CMU-CS-82-111

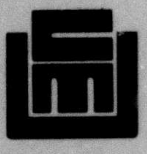
(12)

Architectures and Algorithms for Parallel Updates of Raster Scan Displays

Satish Gupta
Computer Science Department
Carnegie-Mellon University

December, 1981

DEPARTMENT
of
COMPUTER SCIENCE



Carnegie-Mellon University

CMU-CS-82-111

Architectures and Algorithms for Parallel Updates of Raster Scan Displays

Satish Gupta
Computer Science Department
Carnegie-Mellon University

December, 1981

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research is supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Abstract

The frame buffer memory organization is the key to achieving high display performance. Traditional frame buffer designs use the *scan-line organization* which allows several pixels along the length of a scan-line to be updated together. This thesis advocates the *symmetric square organization* which allows the access of square regions of the display. This organization is based on the belief that the regions of the display which are commonly accessed simultaneously are no more likely to be tall and thin than to be short and wide. The square memory organization is studied for representative display applications and is shown to be indeed better than the scan-line organization. Based on the algorithms for these applications, a display design is presented. This design uses one custom designed LSI chip, 64 copies of which are required to implement the frame buffer memory system.

Table of Contents

Acknowledgements

This space has been used by people to thank everybody starting from their dog for licking their stamps to God for creating the universe. I shall restrict myself only to thanking the people who directly helped with my thesis.

The foremost on the list is Bob Sproull, my thesis advisor. Bob was an invaluable advisor and his contributions to my thesis are innumerable. Working with Bob is an immense learning experience and the experience I gained while working with him was extremely helpful in all of my research. As Bob himself has said, advisorship is a tenured position for life, and I intend to hold him to it.

I also received continued help and support from the rest of my thesis committee - Gene Ball, Danny Cohen, Raj Reddy, and Guy Steele. I am sure they set a joint record for the fastest time taken to read a thesis draft. I am extremely thankful to them for their effort.

It is not possible to write a thesis in the area of computer graphics without acknowledging Ivan Sutherland. I owe the initial ideas in this thesis to him.

And finally, the most important contribution to the successful completion of my thesis came from all my friends here at CMU. I believe that friends are the most important part of graduate school, without whom life in graduate school would be impossible. My list of friends would be too long to include in this page, so I will simply say - Thank you, all!

Chapter 1
Introduction

Computer displays encompass different devices like cathode ray tubes, plotters, and printers. But the underlying problems for all such devices can be understood by considering only the CRT. This thesis explores the design space for efficient implementations of CRT displays.

Displays can be built by using either the calligraphic or raster scanning techniques. Calligraphic displays create images by drawing straight lines from point to point.[1] Complicated electronics are used to draw these lines and the displays flicker when too many lines are displayed. By contrast, raster displays create images by scanning the whole display from left to right and top to bottom repeatedly, which is why they are called raster scan displays. The electron beam's intensity is modified appropriately for each point in the display (called a pixel). Hence, raster displays present the image information as intensity samples for each point of the display. This storage is referred to as the *frame buffer*. Unlike calligraphic displays, raster displays offer the advantage of displaying arbitrarily complex, flicker-free images. They also have the advantage of being able to use mass-produced television monitor technology and are cheaper as a result.

This thesis discusses raster scan displays only, and ignores calligraphic displays. CRT terminals used only to display text are also ignored.

1.1. Raster scan displays

In 1970, Ivan Sutherland [Sutherland 70] predicted that raster displays would become a common form of computer output within a very few years. This prediction did not come true for two reasons: memory costs were too high and the raster displays built could not be updated fast enough to be usable for interactive applications. But by now, raster displays are in widespread use.

---

[1] Some calligraphic displays draw curved segments, but they are few and rare.

Table 1-1 shows the cost of random access memory since 1971 for a 512x512 bit-map display, which is a display in which each pixel can be only off or on. Early frame buffers used disks and drums for storage due to the prohibitive cost of random access memory [Terlet 67] [Ophir 68]. When semiconductors became economical, some designs used LSI shift registers [McCracken 75], but since these memories were not very fast, the displays tended to have low resolution. The resolution of Terlet's disk display was 320x192 and McCracken's shift register display had a resolution of 256x256. It was not until recently that random access memory has become cheap enough to be used for inexpensive raster displays.

Year	Cost of Memory
1971	$2500
1973	$1250
1975	$500
1977	$250
1979	$125
1981	$60

Table 1-1: Cost of a 512x512 bit-map display memory over the past decade.

The image on a raster scan display has to be continuously scanned, which requires continual memory access. This process is usually referred to as *screen refresh*. To achieve high resolution, the rate at which the screen is refreshed has to be extremely fast. For example, a 768x1024 display refreshed 60 times per second displays a pixel every 16 nanoseconds. As a result, the memory must access several pixels in parallel in order to keep up with the screen refresh. For example, if the access time for the memory is 256 nanoseconds, at least 16 pixels have to be retrieved in every memory access to maintain the required refresh rate. Idle time during horizontal and vertical retrace would then be used to update the frame buffer.

To achieve high update speeds, several pixels have to be updated simultaneously. This speed is extremely crucial for interactive uses of the display because large amounts of information may have to be changed even for conceptually simple operations. This problem is best illustrated by some examples.

A 768x1024 display can show approximately 6000 characters, occupying an area of approximately 500,000 pixels on the display. Table 1-2 shows the time taken to generate a new screenfull of characters assuming various update times for individual pixels. Frame buffer systems in which the host computer has to update separately every pixel of every character might take 10 μs/pixel, in which case the total time required to generate a new image is unacceptable. To speed up the process,

new characters can be copied into the frame buffer by writing several pixels in parallel. If 16 pixels can be read or written in parallel, then a memory access time of approximately 500 ns allows 16 pixels can be copied in 1 μs, resulting in an update speed of 1/16 μs/pixel.

Update time per pixel	Total time for 6000 characters
10 μs.	5 seconds
1 μs.	.5 seconds
1/16 μs.	30 ms.

Table 1-2: Time taken to update a screen of 6000 characters.

The operation of copying pixels from one part of the frame buffer memory to another is also required when scrolling a window of the frame buffer. For the scrolling of the entire 768x1024 display to appear smooth it should happen in less than one frame time (e.g. 1/60 of a second). This requires an update bandwidth of 45 Megapixels/second.

Typical calligraphic displays can draw several thousand lines during each refresh period. If we assume each vector to be approximately 100 pixels long (about 1/10 of the display), then a frame buffer display should update at least 3 Megapixels/second in order to emulate a vector display. Table 1-2 shows that this cannot be done unless several pixels are updated in parallel.

1.2. Background

Raster display designs started the use of random access memory to store the image only when memory prices declined sufficiently. Some designs refreshed the screen from the computer memory using an elementary peripheral controller [Noll 71] [Thacker 81]. Others implemented the frame buffer as an independent unit with an interface to allow the host computer to read and write image data [Denes 75] [Kajiya 75] [Baskett 76]. These two architectures, known as the *integral frame buffer* and *peripheral frame buffer* respectively, are shown in Figures 1-1 and 1-2 respectively. In the first architecture the memory can be used either by the computer for general and graphics processing or by the video controller for screen refresh. The second architecture separates the computer memory from the frame buffer memory, hence increasing the total bandwidth available to the system. Some designs add a graphics processor between the computer bus and the frame buffer memory to aid in graphics processing (Figure 1-3). Unfortunately, some computers provid such frame buffers on the I/O bus, where graphical activity use the I/O channel and must compete with other channel activities. Some typical functions included in this processor are screen coordinate to memory address translation, line drawing, rectangle filling and character printing.

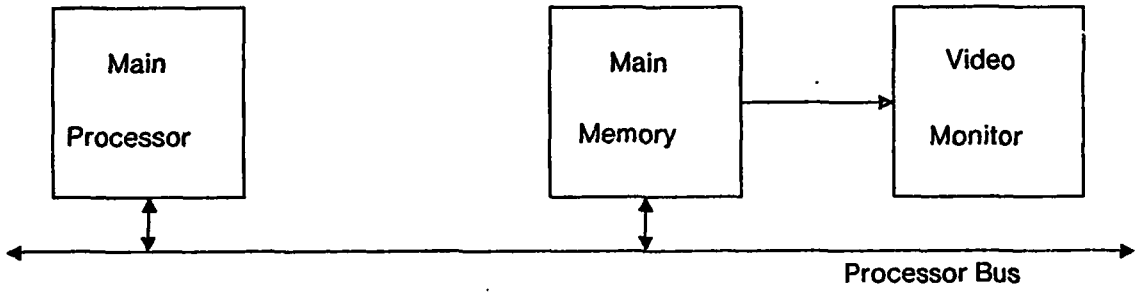


Figure 1-1: Integral frame buffer

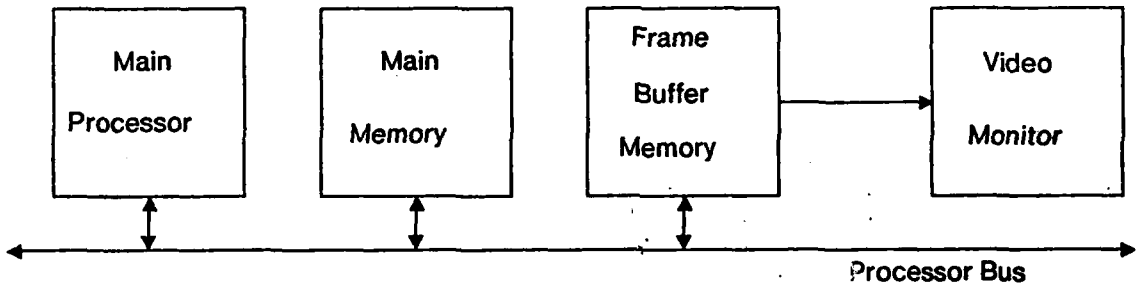


Figure 1-2: Peripheral frame buffer

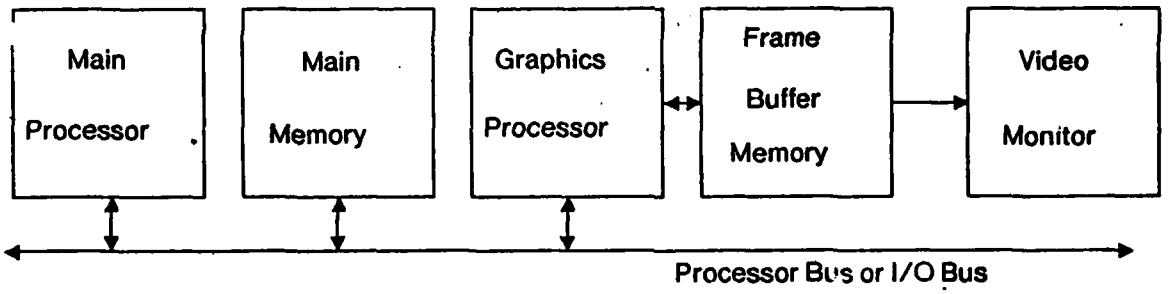


Figure 1-3: Peripheral frame buffer with graphics processor

In all these cases, the memory is organized into words and each word contains more than one pixel [Thacker 81] [Baskett 76] [Bechtolsheim 80]. These pixels are made to lie along a scan line so that the video controller can simply load them into a shift register and shift them out as video for the CRT. As discussed before, this is necessary because of the high data rate required by the refresh. The number of pixels per word is usually 16 or 32. The host computer accesses the frame buffer in between refresh accesses and during the CRT horizontal or vertical retrace.

This kind of memory organization influences the algorithms used to update the image. As mentioned before, it is desirable to update several pixels at a time in order to achieve a fast update of the whole image. The memory organization calls for these pixels to lie along a scan line. This leads to the use of what are commonly known as *scan line algorithms* to create and update images. These algorithms usually generate the image such that successive pixels written lie along a scan line.

The *8x8 display* [Sutherland 81] was designed using a symmetric square organization which allows access to any 8x8 square of the screen. This design is based on the belief that the regions of the display that are commonly accessed simultaneously are no more likely to be tall and thin than to be short and wide. The display is a two-dimensional device, and neither of the axes should be favored by the design. The 8x8 display is intended for applications which copy pixels from one part of the screen to another. This operation is known as BitBlt (BLock Transfer of BITs), and is useful for character generation as well as for scrolling and other rearrangements of images already displayed on the screen.

The 8x8 display provided sufficient evidence that the symmetric square memory organization was indeed a good idea for display memory. An attempt to design a system which would be usable for a larger set of display applications led to this thesis. This thesis presents algorithms for updating squares for a large class of display applications including BitBlt, graphics, and image processing. These algorithms reconfirm that the square organization is indeed an excellent memory organization for efficient updates. The requirements imposed by these algorithms are then used to present a display design for the 8x8 organization.

1.3. Thesis outline

Chapter 2 of this thesis defines the scan-line and square memory organizations. After presenting these two organizations it introduces a staggered square organization which provides access to both the symmetric squares required for updates and scan-line spans convenient for screen refresh.

The idea of having a convenient memory organization for parallel updates can be extended to updating the exact geometry needed by individual applications. For example, the character generator would like to update 5x7 rectangles, the line drawer would like 64x1 for horizontal lines, 1x64 for vertical lines etc. Chapter 3 discusses methods for allowing such indulgences. These methods are not implementable due to current technological limitations, but will certainly provide ways of using future advances in microelectronics. The contents of Chapter 3 are hence not directly related to the main theme of this thesis and may be omitted by the casual reader.

To use fully the capabilities provided by convenient memory organizations, all display applications have to use efficiently the provided memory bandwidth. Chapters 4 and 5 discuss algorithms usable for BitBlt and line drawing. BitBlt is a fairly general operator that provides the ability to move an arbitrarily sized rectangle of the image from one part of the display to another, and can be used to provide most of the higher-level operations required by editing and browsing tasks. BitBlt has been implemented traditionally by using parallel updates in the scan-line organization. Chapter 4 presents these algorithms as well as the algorithms for the square organization. These algorithms are then compared as an aid in deciding which memory organization to use under different circumstances.

Line drawing is the most frequently used graphical primitive. Chapter 5 discusses several algorithms which generate several pixels along a line in each step. These algorithms use the square memory organization and a precomputed set of strokes which can be put together like a jigsaw puzzle to form all possible lines. The strokes are stored in a manner identical to character fonts and are put together using BitBlt. As can be expected, there are tradeoffs, and algorithms that use a larger base set of strokes generate more accurate lines than algorithms that use a smaller set of strokes. The precomputed strokes technique can be extended to trapezoid filling by using precomputed patches. Trapezoids can then be put together to form polygons.

Bit-map graphics suffer from annoying edges called "jaggies". These defects can be removed by using gray-scale pixels to smooth the edges, together with a process known as *anti-aliasing*. Chapters 6 and 7 discuss techniques and algorithms to render anti-aliased graphics. The first of these chapters discusses anti-aliasing techniques. The techniques discussed focus on using table lookups to ease the burden of large computations. These techniques can be implemented very efficiently in frame buffer systems, because extra frame buffer memory can be used to store the tables. Chapter 7 discusses algorithms that use the anti-aliasing techniques to provide high performance graphics. The tasks illustrated are line drawing and trapezoid filling.

The square memory organization is also useful for implementing certain image processing algorithms. Chapter 8 presents algorithms for image processing that are used to present images in more desirable formats. The transformations discussed are translation, scaling, and rotation. While translation over pixel boundaries was the subject of the discussion on BitBlt in Chapter 4, the discussion in this chapter is generalized to translation over sub-pixel distances. Similarly, the discussion on scaling and rotation is generalized to non-integer scaling and non-perpendicular rotation.

Chapter 9 presents the design of a memory system intended for an 8x8 memory organization, which tries to meet the requirements imposed by the various applications and their algorithms. The design uses one custom-designed LSI chip, the specifications of which are presented.

Chapter 2
Display Memory Organization

The frame buffer memory has two primary properties. It can be updated to change the data contained and hence produce new images. It can also be accessed to display the image on output devices. Refresh type output devices such as CRTs require continuous scanning while storage devices like plasma panels require updating only when the image changes. Both the update and the output operations require parallel access of more than one pixel of the image. To provide this parallelism, the frame buffer memory is organized into words, where each word contains more than one pixel. The *memory organization* determines how the pixels in each word map onto the display. If N pixels are accessed in each memory access, then the display can be designed using N random access memory chips, each memory chip being capable of accessing one pixel in each memory access[2]. Another aspect of the organization is the mapping of the location of the N pixels on the display to the addresses to be used for accessing the memory chips.

Although several pixels can be written in parallel we also need the ability to update a subset for occasions when smaller updates are required. This can be done by masking. Masks are N-bit values that specify which of the N memory chips should be written into. They can essentially be used as write-enables for the memory chips, in which case a 0 will enable a memory chip and a 1 will abort the write. Although 2^N masks are possible only a small set is used in practice. The masks typically used are the ones that allow the update of physically contiguous pixels in the N pixel word.

This chapter assumes that the origin of the display coordinate system lies at the top left corner of the screen. The x-coordinates increase towards the right and the y-coordinates increase towards the bottom of the screen. In the mathematical notation used in this chapter, "*" stands for multiplication, "/" for division, and "%" for modulus. The operation $\langle i,j \rangle$ extracts bits i through j of a word, where bit 0 is the least significant bit. The "$\cdot$" operation concatenates bit fields.

---

[2] If each pixel contains more than one bit, then $g*N$ memory chips are used, where g is the number of bits per pixel. If more than one bit could be accessed in parallel from each chip, then fewer chips would be required.

2.1. Scan-line Organization

Conventional display memory organizations map all pixels in each word along a scan line on the display. This organization is motivated by the high data rate required by the video refresh of CRTs. A 768x1024 non-interlaced display refreshes the screen at the rate of 16 ns/pixel. Assuming the memory cycle time to be 250 ns, each memory access has to provide at least the next 16 pixels along the scan-line for the refresh controller to keep up with the data rate required. If this display is implemented with 16 memory chips, then the only time available to update the display is during the horizontal and vertical retrace intervals.

The video controller for a display that uses such a *scan-line organization* can simply load the pixels into an N-pixel shift register and shift them out as the video signal for the CRT. If the same address is given to each of the N memory chips, the N pixels accessed lie along a scan line. With this scheme the N pixels accessed during each memory cycle are forced to be aligned to the N-bit word boundary of the screen (Figure 2-1).

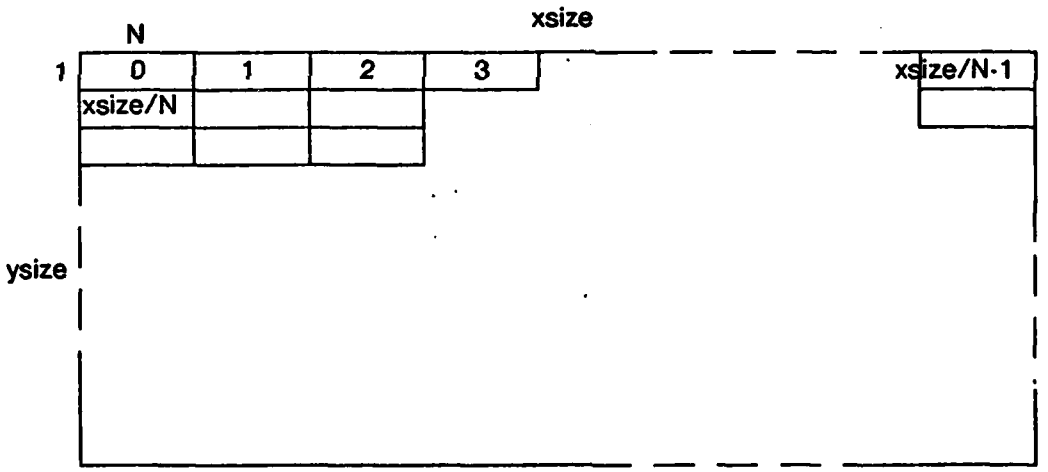


Figure 2-1: Address mapping for the scan-line organization.

If the size of the display is *xsize* by *ysize*, then the size of each RAM chip would be (*xsize ysize*)/N pixels. If $A(x,y)$ is the address used to access a pixel located at (x,y) then a possible address mapping could be

$$A(x,y) := x/N + y(xsize/N).$$

Commonly N, *xsize*, and *ysize* are powers of two which enable the address computation to be

performed by simply using bit-field extractions. As an example, if *xsize* and *ysize* are 1024 each and N is 16 then

$RamSize := 64 \times 1024$ (64K pixels)

and

$A(x,y) := x/16 + 64y$

which can be written as

$A(x,y) := y\langle 9:0\rangle \cdot x\langle 9:4\rangle$

Commercial RAM chips require that the address be split up into two different parts in order to time multiplex the address lines. These two parts are called the Row Address and the Column Address respectively (because they are used as decoders along rows and columns in the memory array). $A(x,y)$ could be split in the following manner to produce 8-bit $RA(x,y)$ and $CA(x,y)$ respectively.

$RA(x,y) := y\langle 1:0\rangle \cdot x\langle 9:4\rangle$
$CA(x,y) := y\langle 9:2\rangle$

In this memory organization, an update of fewer than N pixels has to be performed by using a mask which will disable writing some of the memory chips. Assuming that we want to update only contiguous pixel segments and that this segment can be located at an arbitrary pixel boundary, we need masks originating and ending at every pixel of the N pixel span. Such masks can be specified by $\langle a,b\rangle$, where a specifies the starting position of the memory chips enabled and b specifies the ending position, such that $a > b$ (Figure 2-2). In this representation there exist $N(N-1)/2$ masks. If the value of the enabled mask bit is 0, then each of these masks can be computed as

for $c := 0$ to $N-1$ do
    if $(a \geq c \geq b)$ then $Mask\langle c\rangle := 0$
        else $Mask\langle c\rangle := 1$.

A standard trick is to use only $2N$ masks instead of $N(N-1)/2$. All masks of the form $\langle N-1,b\rangle$ and $\langle a,0\rangle$ are precomputed and $\langle a,b\rangle$ is computed as

$\langle a,b\rangle := \langle N-1,b\rangle$ OR $\langle a,0\rangle$.

This scheme is used in the Lisp machine [Bawden 77].

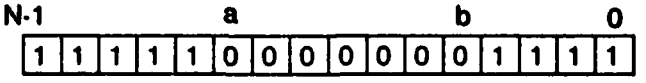


Figure 2-2: Mask which enables the subfield $\langle a,b\rangle$.

A disadvantage of this kind of word organization is that it divides the display into boundaries corresponding to the word boundaries in memory. Updates can take place only on an Nx1 grid aligned to the boundaries of the screen. If it is necessary to write N pixels into the display memory which are not aligned to the N pixel boundary on the screen, the write must be split up into two separate writes to adjoining words of memory. To load an 8x10 character into an 8x1 organized display would probably take 20 memory cycles and 10 only in the rare case of the character being aligned with the word boundary.

The scan-line organization can be modified to provide access to any arbitrary span of N pixels. This is done by addressing different memory chips with separate addresses. As shown in Figure 2-3, an arbitrary span of N pixels can cross only one word boundary and two addresses are sufficient to access the span. In addition, the two addresses differ only by one. If we want to access an N-pixel span whose left edge is located at (x,y), then the addresses used will be $A(x,y)$ and $A(x,y)+1$ and the following procedure can be used to determine the address for each chip. The address used depends upon the offset from the boundary i, and the position of the memory chip which is determined by a identification number c ($0 \le c < N$). All chips in the range ($i \le c < N$) receive the address $A(x,y)$ and all chips in the range ($0 \le c < i$) receive the address $A(x,y)+1$. For $N = 16$, the address computation can be expressed as

$CA := y\langle 9:2\rangle;$
$RA := y\langle 1:0\rangle \cdot x\langle 9:4\rangle;$
$i := x\langle 3:0\rangle;$
if $(c < i)$ then $RA := RA + 1.$

The update algorithms perform better in this case than in the case with aligned boundaries, especially for updating small regions of the screen where splitting the update over the boundary significantly increases the overhead.

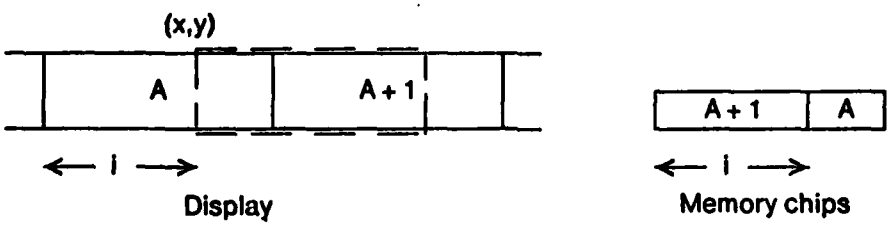


Figure 2-3: Accessing an arbitrarily located Nx1 span

Most image update algorithms perform updates in a manner so that successive operations lie in

close proximity to the previous ones. They can make use of the fact that updating the addresses for the next span is easier than recomputing a new set of addresses. After computing the addresses for a span at (x,y) using the procedure described, the addresses for the span at $(x+N,y)$ can be updated simply by

$$RA(x+N,y) := RA(x,y) + 1.$$

When updates can be aligned to arbitrary pixel boundaries, all smaller updates can be forced such that the left corners of the array of pixels updated correspond to the left corners of the array of pixels addressed. With this restriction we need only N masks for the N possible sizes that can be updated, all of which originate at the left corner. Mask m is characterized by 0s in the leftmost m bits and 1s in the remaining bits. This mask can be used with a one-to-one correspondence to the write-enables of the N memory chips only when the corner of the span being accessed abuts the Nx1 grid boundary. In the case where the span being updated has an offset i from the boundary of the grid, the left bit of the mask is used as the write-enable for the ith memory chips, the next bit for the $i+1$ memory chips and so on. This additional complication can be solved in either of two ways. We could either precompute masks for each value of i, which again requires us to have N^2 masks, or we could use the N masks and align them to the memory chips depending upon the the value of i. This alignment can be performed by rotating the mask to the right by a rotate count of i and will be the subject of a more detailed discussion in Chapter 4.

2.2. Symmetric Organization

The scan line organization imposes an inherent asymmetry on update operations: horizontal updates are easier than the vertical ones. The asymmetry is easily seen in the line drawing example. A horizontal line can be drawn very quickly but only one pixel of a vertical line can be drawn in each memory cycle. It is for these reasons that the designers of the 8x8 display [Sutherland 81] chose a symmetric 8x8 organization. The 8x8 display can read or write 64 pixels in each memory cycle; the 64 pixels read or written lie on an 8x8 square on the screen. The complexity of line drawing is now symmetric with respect to the x and y axes of the screen. This symmetric organization effectively provides a lower access time for image generation. Successive pixels generated by most image generators (we have seen characters and lines already) lie close to each other in any direction. This organization allows them to be updated together, which effectively provides them a low average access time.

An MxM memory organization requires M^2 memory chips. Each memory chip will provide one

pixel for each MxM square of the screen and, if the memory array is viewed as an MxM square, then there is a one-to-one correspondence between the memory chips and the pixels of the screen (see Figure 2-4). The top leftmost bit of each MxM square on the screen comes from the top leftmost chip of the memory array, the top rightmost bit from the top rightmost chip, the bottom leftmost bit from the bottom leftmost chip, and so on. If each chip is identified by a column number and a row number in the array, then point (x,y) on the screen will be located in chip $(x\%M, y\%M)$.

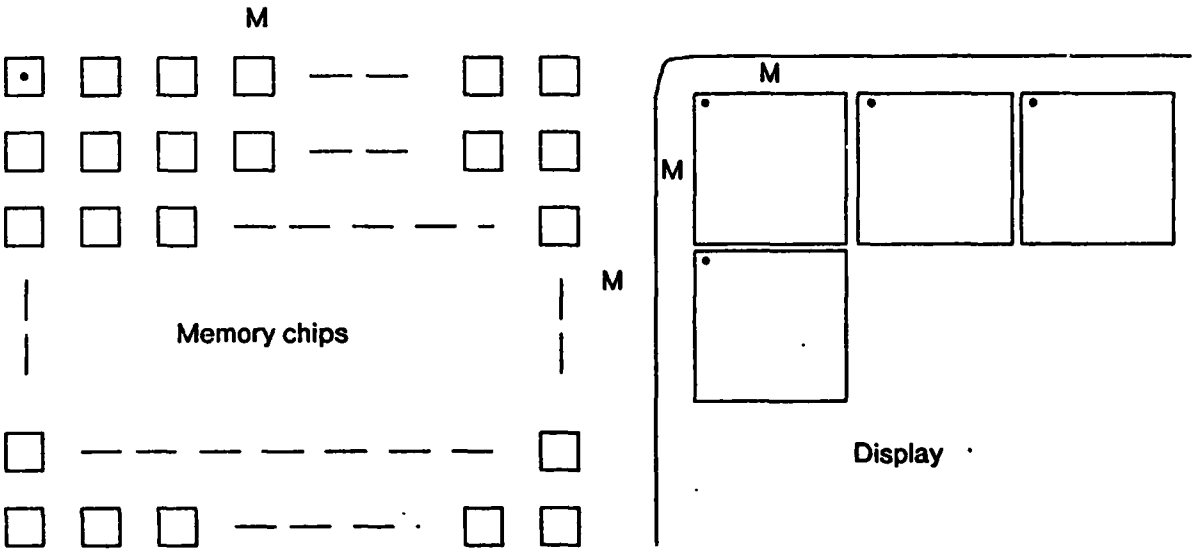


Figure 2-4: Symmetric Memory Organization.
Marked pixels are mapped to the top left memory chip.

The address provided to the memory will be determined by the position of the MxM square accessed. All MxM squares on the screen can be identified by a column number and a row number (there are $xsize/M$ columns and $ysize/M$ rows). These column and row numbers can easily provide the column and rows addresses required by the memory. The column number of the MxM square is used as the column address to the memory chips and the row number is used as the row address. Hence point (x,y) on the screen will be located in chip $(x\%M, y\%M)$ at address $(x/M, y/M)$ where x/M is the column address and y/M is the row address.

The addressing scheme described has the same boundary problem that we discussed before. It does not allow access to an arbitrarily positioned MxM square on the screen in one memory cycle. However, the 8x8 display allows an arbitrary MxM square to be accessed in one memory cycle by providing separate addresses to different parts of the memory array. As shown in Figure 2-5, if we

want to access the MxM square positioned at (x,y), then the memory chips in region A will be addressed by $(x/M,y/M)$, chips in B by $(x/M+1,y/M)$, chips in C by $(x/M,y/M+1)$, and chips in D by $(x/M+1,y/M+1)$. The square we want to access is offset from the boundaries by $i\,(= x \bmod M)$ and $j\,(= y \bmod M)$. If the memory chips are identified by (c,r) (where c is the column number and r is the row number), then Region A in Figure 2-5 is identified by the memory chips in the range $(i \le c \le M, j \le r \le M)$, Region B by $(0 \le c < i, j \le r \le M)$, Region C by $(i \le c \le M, 0 \le r < j)$, and Region D by $(0 \le c < i, 0 \le r < j)$. This computation can be represented procedurally as

$CA(x,y) := x / M;$
$RA(x,y) := y / M;$
$i := x \,\%\, M;$
$j := y \,\%\, M;$
if $(c < i)$ then $CA := CA + 1;$
if $(r < j)$ then $RA := RA + 1.$

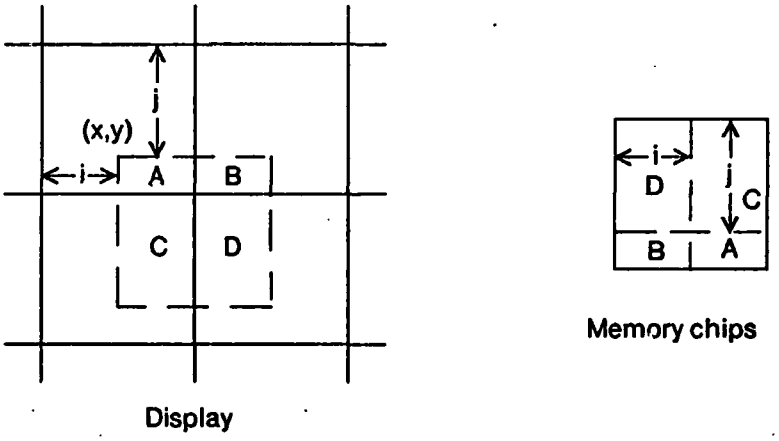


Figure 2-5: Accessing an arbitrarily aligned MxM square

These addresses are easily updated for squares in close proximity. The addresses for the square at $(x+M,y)$ can updated simply by

$$CA(x+M,y) := CA(x,y) + 1.$$

Similarly the addresses for the square at $(x,y+M)$ can be updated by

$$RA(x,y+M) := RA(x,y) + 1.$$

Masking in the two-dimensionally symmetric organization is a two-dimensional version of the masking problem in the scan-line organization. We now need the ability to update rectangles smaller

than the MxM square. As in the scan-line organization we can restrict these rectangles to be updated such that the upper left corner of the rectangle corresponds to the upper left corner of the MxM square being updated. With this restriction we need only M^2 masks for all possible rectangles. Mask (p,q) is defined by 0s in the top left pxq rectangle and 1s in the rest of the square. These two-dimensional masks can be obtained by ORing two one-dimensional masks. The x-directional parameter p is used to obtain a mask with 0s in the left p columns and the y-directional parameter q similarly obtains a mask with 0s in the top q rows. The two masks are ORed to give the desired rectangular mask.

As in the case of the scan-line organization, these masks have to be aligned to move the bits to the correct memory chip where they can be used as the write enables. This alignment requires a two-dimensional rotation that will be discussed in Chapter 4.

2.3. Staggered square organization

The unconventional square organization makes raster-scanned screen refresh harder than in the scan-line organization. The problem arises from the fact that only M pixels out of the M^2 accessed in one memory access lie along the scan-line being refreshed. The choice is either to use just these M pixels and ignore the rest which would result in a very low memory bandwidth utilization for screen refresh or to store all the pixels read into a separate buffer and use the buffer for refreshing successive scan lines. The 8x8 display provides two buffers each capable of holding 8 scan lines each. One of the buffers is used to refresh the screen while the other buffer is being filled with the next 8 scan lines from the memory.

A staggered organization of the form shown in Figure 2-6 can be used to provide access to both the symmetrical squares required by the update operations and the horizontal spans required for the screen refresh. Successive squares along the x-direction are shifted up by one line. The edge conditions on the top and bottom edges are wrapped around. This staggering still allows the access of any arbitrarily positioned MxM square, while also allowing the access of any M^2x1 span of pixels to ease the screen refresh. Figure 2-6 also shows how this organization allows access to horizontal spans. The first M pixels of the span are stored in the top row of memory chips, the next M pixels are stored in the second row, and so on until the final M pixels of the span are stored in the bottom row of memory chips. The fact that all the pixels in the span are stored in different memory chips allows them to be accessed in one memory cycle.

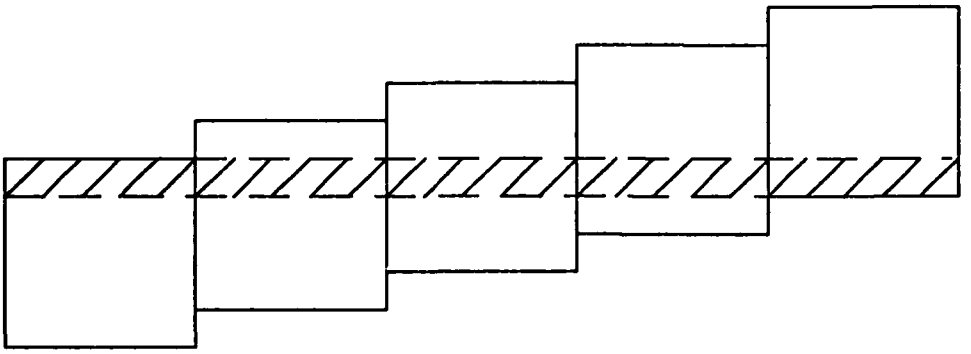


Figure 2-6: Staggering 5x5 squares to access horizontal spans.

Figure 2-7 shows the memory mapping of this staggered arrangement. The pixel at (x,y) is located in the memory chip $(x\%M,(y+x/M)\%M)$ and is addressed by $(x/M,(y+x/M)/M)$. This mapping is similar to the unstaggered mapping with the replacement of y by $(y+x/M)$. This replacement represents the staggering by adding to the value of y the displacement added by the stagger which depends upon the x position.

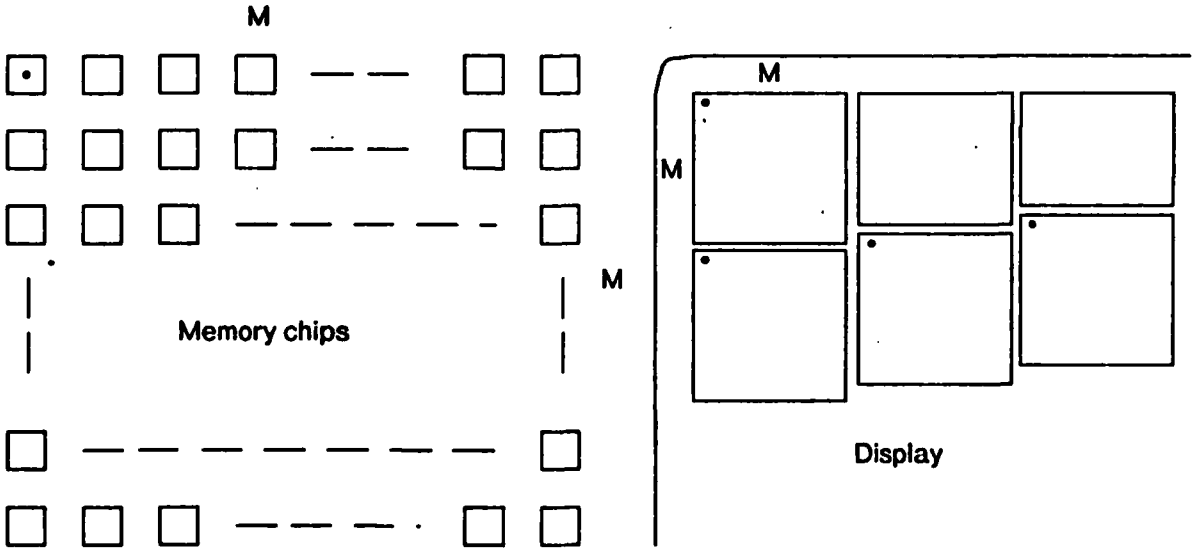


Figure 2-7: Staggered square organization.
Marked pixels are mapped to the top left memory chip.

2.3.1. Addressing squares

Four different addresses are required to access an arbitrarily positioned MxM square. The situation is shown in Figure 2-8 which shows a MxM square positioned at (x,y). This square is offset from the boundaries by $i\,(= x/M)$ and $j\,(= (y+x/M)/M)$. Region A is identified by the chips in the range $(i \le c \le M, j \le r \le M)$, and is addressed by a column address (CA) of (x/M) and row address (RA) of $(y+x/M)/M)$. Region B is identified by $(0 \le c < i, j+1 \le r \le M)$ and addressed by $(CA+1,RA)$. Region C is identified by $(i \le c \le M, 0 \le r < j)$ and addressed by $(CA,RA+1)$. Region D is identified by $(0 \le c < i, 0 \le r < j+1)$ and addressed by $(CA+1,RA+1)$. The addresses can be procedurally computed in the following manner:

$CA := x/m;$
$RA := (y+x/m)/m;$
$i := x \,\%\, m;$
$j := (y+x/m) \,\%\, m;$
if $(c < i)$ then $j := j + 1;$
if $(c < i)$ then $CA := CA + 1;$
if $(r < j)$ then $RA := RA + 1.$

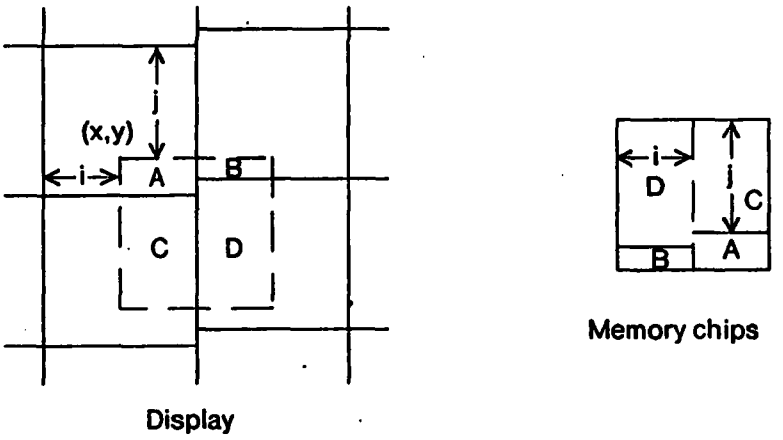


Figure 2-8: Accessing an arbitrarily aligned MxM square

Incrementing the addresses for accessing the square at $(x,y+M)$ is simply

$$RA(x,y+M) := RA(x,y) + 1.$$

The problem with incrementing the addresses for accessing the squares at $(x+M,y)$ is that the boundary marking the squares changes its position and hence it is not sufficient to merely increment the column address. The situation is illustrated in Figure 2-9, which marks the memory chips that

have to increment their row address. These memory chips are identified in the range ($i \leq c < M$, $r = j+1$) and ($0 \leq c < i$, $r = j$). The value of j has already been modified to $j+1$ when $c < i$. Thus we can combine the two sets into ($0 \leq c < M$, $r = j$), and can procedurally represent the incrementation of the addresses for $(x+M,y)$ as

$CA(x+M,y) := CA(x,y) + 1;$
if $(r = (j \% M))$ then $RA(x+M,y) := RA(x,y) + 1.$

Further increments along the x-direction can be handled in two ways. We can either continue to modify the value of j so that continued increments can be done in exactly the same way as the first one. The shift of boundaries merely increments the value of j, and that is all that has to be done to be able to continuously increment the addresses. The incrementing procedure can now be written as

$CA(x+M,y) := CA(x,y) + 1;$
if $(r = (j \% M))$ then $RA(x+M,y) := RA(x,y) + 1.$
$j := (j + 1) \% M.$

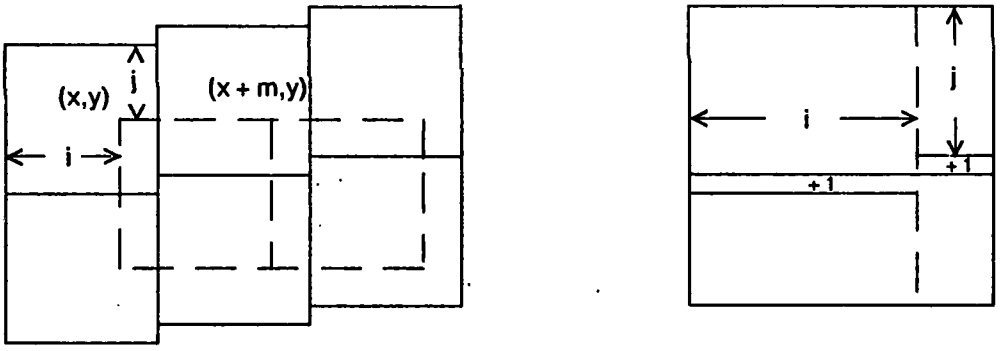


Figure 2-9: Row addresses to be incremented when square to be accessed is moved horizontally

The second method for continued increments along the x-direction assumes the presence of an NxN array, $RAInc$. The row addresses of the memory chips are incremented by this array to compute the new row addresses. The $RAInc$ array is then updated so that further increments can be done in a similar manner. This increment for the initial placement is computed as

if $(r = (j \% M))$ then $RAInc[c,r] := 1$
else $RAInc[c,r] := 0.$

Since further udpates require the value of j to be incremented by 1, the $RAInc$ array can be updated

by circularly shifting it in the y-direction by a single step. The address updating procedure in this case is

$$CA(x+M,y) := CA(x,y) + 1;$$
$$RA(x+M,y) := RA(x,y) + RAInc[c,r];$$
$$RAInc[c,r] := RAInc[c(M+r-1)\%M].$$

2.3.2. Addressing horizontal spans

We shall assume that when accessing a horizontal span whose left corner is located at (x,y), x is a multiple of M^2. This assumption is justifiable when this mode of memory access is used only for screen refresh. The leftmost M pixels of this span are located in chips ($0 \le c < M$, $y\%M$) and are addressed by a column address of (x/M) and a row address of $(y+x/M)/M$. Chips in other rows shall be addressed by a column address that is a sum of x/M and the difference between the chip row number and $(y\%M)$. Two row addresses are used because the span is likely to cross a stagger boundary (the one in Figure 2-7 does not). The address for a chip (c,r) can be computed as

$$CA := x/M + (M + r - (y \% M)) \% M;$$
$$RA := (y + x/M) / M;$$
$$\text{if } ((y \% M) > r) \text{ then } RA := RA + 1.$$

This address is easily incremented when the span at $x := x + M^2$ is to be accessed. It can be done as

$$CA(x+M^2,y) := CA(x,y) + M;$$
$$RA(x+M^2,y) := RA(x,y) + 1.$$

2.4. Conclusion

The frame buffer memory organization is the key to achieve high display performance. The effective utilization of the memory organization depends on an efficient address computation. The efficiency of the address computation can be increased by simple incremental algorithms to access parts of the display in close proximity of the previous access. This chapter presents the *scan-line* and the *square* orgnization, both of which have simple address computation algorithms. The square organization is more suited for update algorithms but suffers from an expensive screen refresh implementation. The *staggered square* organization allows the access of squares for updates and scan-line spans for screen refresh, but requires expensive address computation algorithms.

Chapter 3
Flexible Memory Organizations [3]

The previous chapter discussed one memory organization which provides two kinds of accesses; it provides the ability to access squares for the convenience of update algorithms and it can also be used to access horizontal spans for the raster scanning required by screen refresh. The flexibility of accessing the memory in the form desired by each task effectively increases the total memory bandwidth and the performance of the display system. The screen refresh task requires accessing the largest possible horizontal spans. The claim of this thesis is that if all update tasks had the choice of one memory organization they would choose the square organization discussed in the previous chapter.

If update algorithms can use different geometries to access the display memory, they can choose one which minimizes the number of memory cycles used. For example, if the memory organization provides both horizontal and vertical spans, then a line drawing algorithm could use the horizontal access for flat lines and the vertical access for the steeper lines. The flexibility can be extended to provide access to rectangles with all different sizes and shapes, in which case the line drawing algorithm would merely choose a size depending upon the length of the line and a shape depending upon the slope of the line and update the line in one memory access. The maximum size allowed determines the maximum bandwidth possible and also the cost of the system. Allowing for *all* different sizes requires parallel access to *all* the pixels of the display and hence cannot make use of any random access memory technology. If the maximum size were restricted to N, and the shapes were restricted to be rectangular, then the most flexibility is provided by an organization which allows the access of all rectangles with sizes $1 \times N$, $2 \times \lfloor N/2 \rfloor$, $3 \times \lfloor N/3 \rfloor$, ..., $\lfloor N/2 \rfloor \times 2$, and $N \times 1$. Such an organization is useful for the following reasons:

---

[3] This chapter is not directly related to the main theme of this thesis and may be omitted by the casual reader. Dan Hoey provided substantial assistance to obtain the results in Section 3.3.

1. The update algorithms will perform better when they attempt to update the screen using the minimum number of memory cycles. For example, a line-drawing algorithm can now select a rectangle depending upon the slope of the line (Figure 3-1).

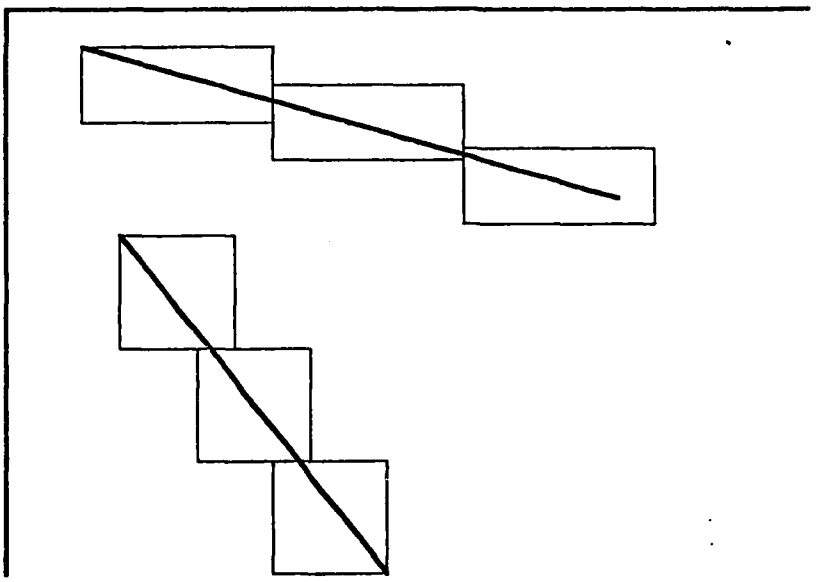


Figure 3-1: Line-drawing algorithms can choose rectangular shapes depending upon the slope of lines.

2. The same frame buffer memory can be used to configure different types of displays with different resolutions and gray-levels. If we want a 1024x1024 bit-map display which allows the access of 8x8 squares or 64x1 spans, or a 512x512 4 bit/pixel display which allows the access of 4x4 squares or 16x1 spans. If the gray-scale pixel maps onto the bit-map display as a 2x2 square, then the memory organization has to provide access to 8x8, 64x1, and 32x2 rectangles.

3. We can drive different output devices from the same frame buffer. If the frame buffer memory provides both the horizontal and the vertical accesses then we can simultaneously drive a CRT which scans horizontally and a hard copy printer which may scan vertically.

The organizations discussed in the previous chapter had the property that all the memory chips in every organization could provide useful data in each memory access. This implied that the area of display accessed was equal to the number of memory chips in the system, with a memory chip defined to be a piece of random access memory that can access one pixel in each memory cycle. The scan-line organization that allows the access of N pixels along a scan line uses N memory chips; the square organization that accesses MxM squares uses M^2 memory chips; and the staggered organization that accesses both MxM squares and M^2x1 spans and also uses M^2 memory chips.

When we increase the flexibility of accessing the display memory system, we will see that we need more memory chips than the maximum size of the display area that can be accessed in each memory cycle. Hence some of the memory chips will not access useful data in each memory access, although the set of chips in use is different for different accesses. An attempt should be made to use the minimum number of memory chips for the organization used although implementation simplicities might favor the choice of a more expensive scheme.

Each memory organization can be defined by the mapping of each display pixel to the memory chip number and represented by a function *ChipNumber*(x,y), where x and y are the coordinates of the pixel on the display. Given the memory organization function, two functions define the computations that have to be performed for each memory chip during every memory access. The first of these two functions computes the address for each memory chip from the specifications of the rectangle to be accessed. The second function computes the mask when only a subset of the rectangle to be accessed is to be written into. The addressing function can be represented as *Address*(x,y,dx,dy,c) which computes the address for the chip numbered c when the rectangle with its top left corner at (x,y) of size (dx,dy) is being accessed. The address function is often broken up into two parts representing the row and column addresses of the memory chips and can be specified as *RowAddress*(x,y,dx,dy,c) and *ColAddress*(x,y,dx,dy,c). The masking function is represented as *Mask*(x,y,dx,dy,mdx,mdy,c) and computes the mask for the chip numbered c when the rectangle specified by x,y,dx, and dy is being accessed and only the top left subrectangle of size (mdx,mdy) is to be written into. This function will return a zero for the chips that have to be enabled and one for the others. The following table contains these functions for the scan-line, square, and the staggered organizations. The comparisons return one if they succeed and zero otherwise; they represent the address increment required by the memory chips that cross word boundaries.

Scan-line organization for a display sized 1024x1024.
Memory chips are numbered $0 \le c \le 15$.

$i := x\langle 3:0\rangle$
$ChipNumber(x,y) := i$
$RowAddress(x,y,16,1,c) := y\langle 1:0\rangle \cdot x\langle 9:4\rangle + (c < i)$
$ColAddress(x,y,16,1,c) := y\langle 9:2\rangle$
$Mask(x,y,16,1,mdx,1,c) := ((16 + c - i) \% 16) \ge mdx.$

Symmetric square organization for a display sized 1024x1024.
Memory chips are numbered (c,r) where $(0 \le c \le 7)$ and $(0 \le r \le 7)$.

$i := x \% 8; j := y \% 8$
$ChipNumber(x,y) := (i, j)$
$ColAddress(x,y,8,8,c,r) := x/8 + (c < i)$
$RowAddress(x,y,8,8,c,r) := y/8 + (r < j)$
$Mask(x,y,8,8,mdx,mdy,c,r) := (((8 + c - i) \% 8) \ge mdx)$
$\quad \vee (((8 + r - j) \% 8) \ge mdy).$

Staggered square organization for a display sized 1024x1024.
Memory chips are numbered (c,r) where $(0 \le c \le 7)$ and $(0 \le r \le 7)$.

$i := x \% 8; j := (y + x/8) \% 8$
$ChipNumber(x,y) := (i, j)$
$ColAddress(x,y,8,8,c,r) := x/8 + (c < i)$
$RowAddress(x,y,8,8,c,r) := (y + x/8)/8 + (r < (j + (c < i)))$
$Mask(x,y,8,8,mdx,mdy,c,r) := (((8 + c - i) \% 8) \ge mdx)$
$\quad \vee (((8 + r - (j + (c \ge i))) \% 8) < mdy).$

When x is multiple of 64

$ColAddress(x,y,64,1,c,r) := x/8 + (8 + r - (y \% 8)) \% 8$
$RowAddress(x,y,64,1,c,r) := (y + x/8)/8 + ((y \% 8) > r).$

Table 3-1

The rest of the chapter discusses some more flexible organizations, and describes their characteristic functions. The organizations restrict themselves to accessing rectangular geometries.

3.1. Row-Column organization

The first unconventional organization is one that allows parallel access of any adjacent set of N pixels along rows or columns. This organization has potential application in displays which drive two different output devices, one of which is scanned along rows and the other scanned along columns. Update algorithms behave symmetrically along either axes although diagonal update can be performed only one pixel at a time.

This organization requires that all pixels in every adjacent set be located in different memory chips. Figure 3-2 shows one mapping from pixel location to memory chip number for $N = 4$. This

0 (0,0)	1 (0,0)	2 (0,0)	3 (0,0)	0 (2,0)	1 (2,0)	2 (2,0)	
1 (0,1)	2 (0,1)	3 (0,1)	0 (0,1)	1 (2,1)	2 (2,1)	3 (2,1)	
2 (1,0)	3 (1,0)	0 (1,0)	1 (1,0)	2 (3,0)	3 (3,0)	0 (3,0)	
3 (1,1)	0 (1,1)	1 (1,1)	2 (1,1)	3 (3,1)	0 (3,1)	1 (3,1)	
0 (0,2)	1 (0,2)	2 (0,2)	3 (0,2)	0 (2,2)	1 (2,2)	2 (2,2)	
1 (0,3)	2 (0,3)	3 (0,3)	0 (0,3)	1 (2,3)	2 (2,3)	3 (2,3)	
2 (1,2)	3 (1,2)	0 (1,2)	1 (1,2)	2 (3,2)	3 (3,2)	0 (3,2)	
3 (1,3)	0 (1,3)	1 (1,3)	2 (1,3)	3 (3,3)	0 (3,3)	1 (3,3)	

Figure 3-2: Row-Column organization.

mapping staggers the chip numbers in consecutive rows but still requires only N memory chips. For this organization

$$ChipNumber(x,y) := (x+y) \% 4$$

Figure 3-2 also shows in parentheses the row and column addresses required to address the memory chips. The addressing mechanism is symmetric over 4x4 squares but asymmetric within each 4x4 square. It addresses the pixel (x,y) with a column address of $(2(x/4) + (y\%4)/2)$ and a row address of $(2(y/4) + y\%2)$. This addressing scheme is symmetric in a way because the column addresses use only higher order bits of x and row addresses use only higher order bits of y but both of them use one lower order bit of y each. The addressing and masking functions are

$$i := (x + y\%4) \% 4;\ j := (y + x\%4) \% 4$$

$$ColAddress(x,y,4,1,c,r) := 2(x/4) + (y\%4)/2 + 2(c < i)$$
$$RowAddress(x,y,4,1,c,r) := 2(y/4) + y\%2$$
$$Mask(x,y,4,1,mdx,1,c,r) := ((4 + c - i)\%4) \geq mdx$$

$ColAddress(x,y,1,4,c,r) := 2(x/4) + (y\%4)/2$
$RowAddress(x,y,1,4,c,r) := 2(y/4) + y\%2 + 2(c < j)$
$Mask(x,y,1,4,1,mdy,c,r) := ((4 + c - j)\%4) \geq mdy.$

3.2. Row-Column-Square organization

The previous chapter presented a staggered organization that could be used to access both squares and horizontal spans; the same technique can be used to access both squares and vertical spans. The previous section presented an organization that can be used to access both horizontal and vertical spans. All these organizations use the same number of memory chips as the area of the rectangles being accessed. But when we try to access squares, horizontal and vertical spans of area M^2 we need M^2+1 memory chips. Figure 3-3 shows one possible memory mapping for $M = 4$ that is used to access 4x4 squares and both 16x1 and 1x16 spans. This memory mapping relies on the fact that M is relatively prime to M^2+1. The *ChipNumber* function increments by 1 along the x-direction and by M along the y-direction. For $M=4$, this function is

$ChipNumber(x,y) := (x+4y)\%17.$

Assume that the pixel at (x,y) is addressed by $(x/17)\cdot y$. There is no preferred way for splitting this into row and column addresses and hence we shall leave the address undivided.

Horizontal spans can be addressed using the following functions -

$i := x\%17$
$Address(x,y,17,1,c) := (x/17 + (c < i))\cdot y$
$Mask(x,y,17,1,mdx,1,c) := (17 + c - i)\%17 \geq mdx.$

Accessing vertical spans can be eased by the presence of a two dimensional *vertydist*[a][b] table which is indexed by chip numbers in both the dimensions. If the value of a is the chipnumber of the top of a span then the table will give the vertical distance to the next lower pixel with a chipnumber equal to b. So for example, *vertydist*[0][8] = 2, and *vertydist*[16][2] = 5. Given the presence of such a precomputed table the addressing functions for vertical spans are

$d := ChipNumber(x,y)$
$Address(x,y,1,17,c) := (x/17)\cdot(y + vertydist[d][c])$
$Mask(x,y,1,17,1,mdy,c) := vertydist[d][c] \geq mdy.$

Accessing squares can be done in a manner similar to the one used to access vertical spans with the difference of requiring two precomputed tables both which are similar to the *vertydist* table. The first one is *squarexdist* which gives the x-distance from the corner memory chip which is the first

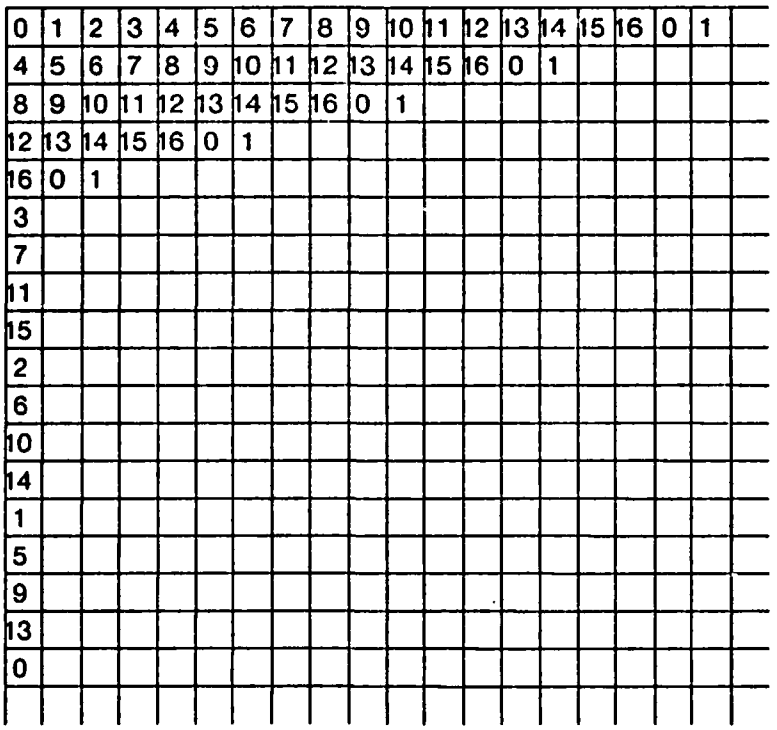


Figure 3-3: Row-Column-Square organization

parameter to the chip which is the second parameter. The second table is *squareydist* whose result in the y-distance in a manner similar to the first table. As an example *squarexdist*[0][9] = 1, *squareydist*[0][9] = 2, *squarexdist*[14][3] = 2, and *squareydist*[14][3] = 1. The addressing functions for such a formulation are

$$d := ChipNumber(x,y)$$
$$Address(x,y,4,4,c) := ((x + squarexdist[d][c])/17) \cdot (y + squareydist[d][c])$$
$$Mask(x,y,4,4,mdx,mdy,c) := (squarexdist[d][c] \geq mdx) \vee (squareydist[d][c] \geq mdy).$$

This technique of using precomputed tables can be extended to general rectangular accesses by using two precomputed tables for each geometry to be accessed.

3.3. General rectangular organization

We are now going to extend the memory organization to allow the access of every rectangle on the display with an area of less than or equal to N in one memory access. The key question is to determine the minimum number of memory chips that are required to implement any such organization. The most obvious implementation puts pixels spaced N apart in both directions in the same memory chips (Figure 3-4). The area enclosed by such pixels is N^2 and since all these pixels have to be located in different memory chips the number of memory chips required is N^2. An improvement to this implementation is shown in Figure 3-5. The repetition pattern in this implementation is trapezoidal instead of being square and requires only $N^{3/2}$ memory chips.

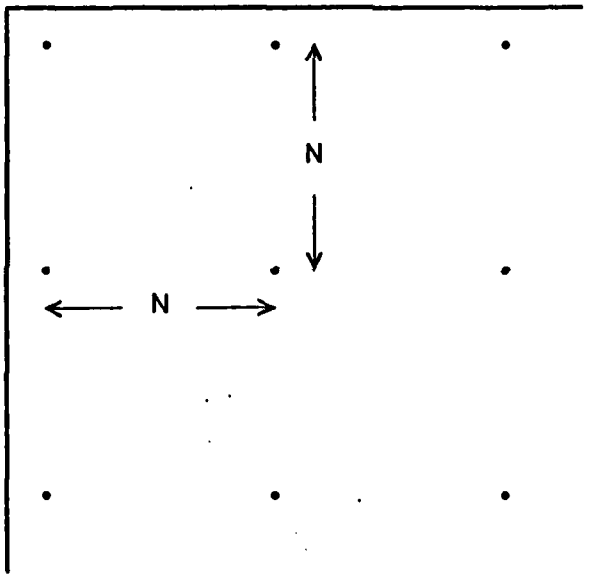


Figure 3-4: Generalized rectangles using N^2 chips

In order to improve the chip count we look at the bounds of one pixel, in the sense that once this pixel has been placed in any memory chip, all pixels enclosed within the boundary cannot be in the same memory chip (Figure 3-6). For simplicity we assume that the pixel whose bounds we are considering is located at (0,0). The curves on the boundary are hyperbolas because the size of the rectangles bounds the outlines to $(x\,y) = N$. Suppose we can place one pixel on the boundary located at (a,b) in the same memory chip as the pixel in the center. Symmetry would dictate that we can also place the pixels at $(-b,a)$, $(-a,-b)$, and $(b,-a)$ in the same memory chip. To access all rectangles of size N, no pixel should be within the boundary of another pixel in this set. This constraint can be stated as

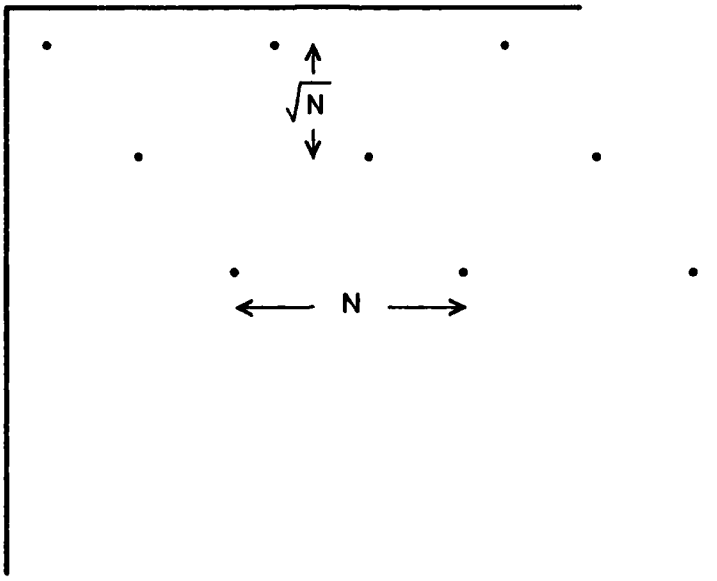


Figure 3-5: Generalized rectangles using $N^{3/2}$ chips

$a\,b \leq N$ and $(a-b)(a+b) \leq N$.

The first inequality can be rewritten as

$b \leq N/a$,

and substituting this value of b into the second inequality results in the following quadratic inequality.

$$a^4 - a^2N - N^2 \leq 0.$$

This inequality has two solutions

$a^2 = N\varphi$
and $a^2 = N/\varphi$
where $\varphi = (\sqrt{5}+1)/2$.

b will have complementary solutions and the solution pair is hence $(\sqrt{N\varphi}, \sqrt{N/\varphi})$ and $(\sqrt{N/\varphi}, \sqrt{N\varphi})$. Figure 3-7 shows the pixels contained in one memory chip when this pattern is repeated. The total number of memory chips is the area enclosed by four such pixels which is equal to $a^2 + b^2 = N(\varphi + 1/\varphi) = \sqrt{5}\,N$. This implementation hence requires only a linear number of memory chips.

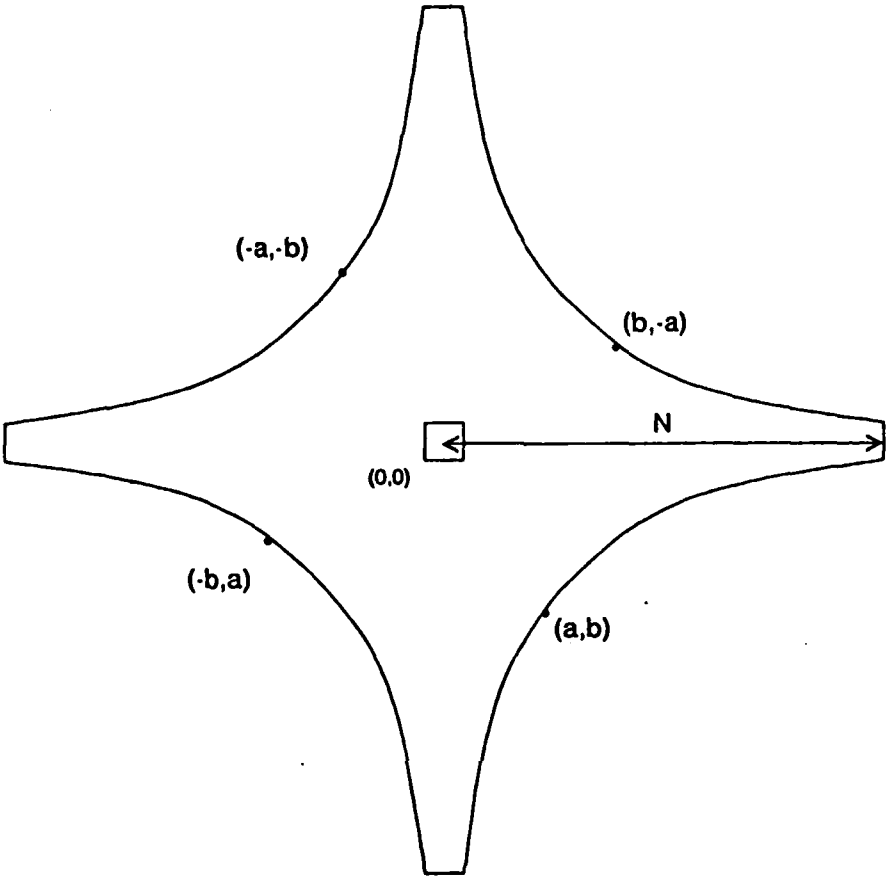


Figure 3-6: Bounds of a pixel when all rectangles are allowed to be accessed. No pixels within the boundary can be in the same chip as the central pixel.

In order to complete this discussion we have to prove that when the proposed pattern is repeated, no pixels are within the bounds of any other pixel. To prove this, we shall consider any arbitrary pixel in the pattern and show that it does not lie within the bounds of the pixel at (0,0). The coordinates of all pixels in a pattern are formed by a certain number of moves in the $(a,N/a)$ direction and a certain number of moves in the $(-N/a.a)$ direction. If the number of moves in the first direction is p and the number of moves in the second direction is q then the coordinates of such a point are $(pa-qN/a,pN/a+qa)$. In order that this point lie outside the bounds of the origin we must have

$$(pa-qN/a)(pN/a+qa) \geq N$$

or

$$p^2N - q^2N + Npq(a^2-1/a^2) \geq N$$

Since $a^2-1/a^2 = 1$, we have

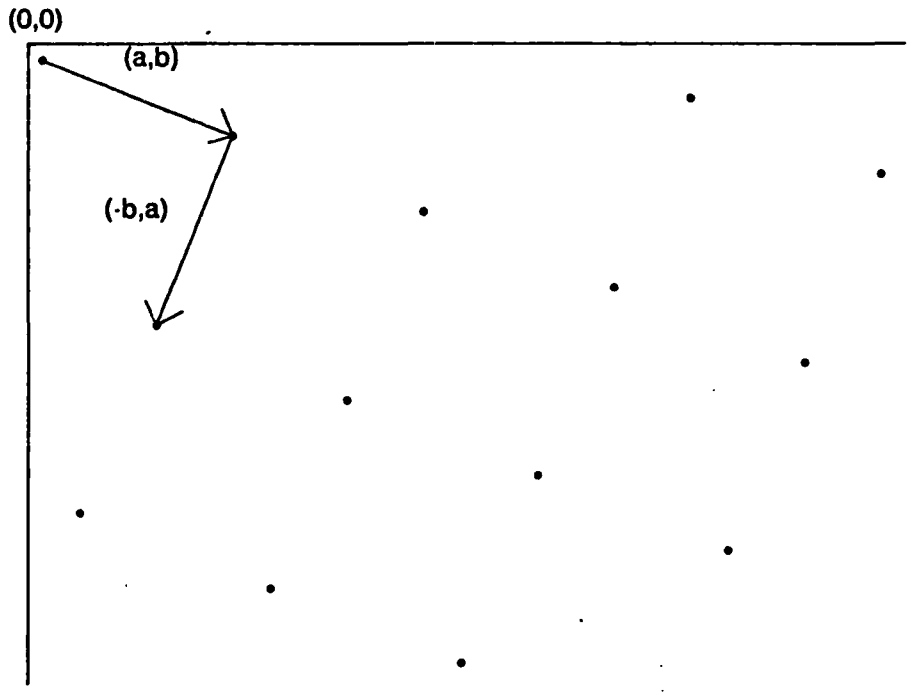


Figure 3-7: Pixels contained in one chip in the general rectangular organization

$$N(p^2 - q^2 + pq) \geq N$$

which is true because $p^2 - q^2 + pq = 0$ has no integer solutions.

The above formulation provides a non-integer bound on the number of memory chips required. The problem is, however, an integer problem and the values used in the mapping have to be integers. Let us consider the repetition pattern in Figure 3-7 assuming that a and b are integers. In order for (a,b) to be outside the bounds of (0,0) we must have $(a+1)(b+1) > N$ (ab could be less than N because there might not be a closer way to factor N). Also in order for (a,b) to be outside $(b,-a)$ we must have $(a-b+1)(a+b+1) > N$. The smallest values of a and b to satisfy this equality can be used in the organization. Figure 3-9 plots the number of memory chips required for different rectangular sizes. The straight line represents $\sqrt{5}N$. The figure confirms our $\sqrt{5}N$ approximation for the number of chips required to implement a general rectangular organization.

For $N = 16$, $a = 5$ and $b = 2$ are the smallest integers to satisfy the two inequalities. This memory organization hence requires 29 ($= a^2 + b^2$) memory chips to enable the access of all rectangles with an area less than or equal to 16. The pixel located at (5,2) will be contained in the same memory chip as

the pixel located at (0,0). In Figure 3-7 we had observed that all pixels spanned by moves in the (a,b) and $(-b,a)$ directions are contained in the same memory chip. Hence all pixels located at $(5p-2q,2p+5q)$, for all integer values of p and q, are placed in the same memory chip as the pixel located at (0,0). The next pixel in the first row placed in the same memory chip as the pixel at (0,0) is located at (29,0), when $p = 5$ and $q = -2$. We can hence place all pixels from (0,0) to (28,0) in the 29 different memory chips resulting in a memory mapping shown in Figure 3-8. The closest pixel in the second row which is in the same memory chip as the pixel at (0,0) is located at (17,1) corresponding to when $p = 3$ and $q = -1$. To preserve a unit increment in chip numbers in all rows we can start by placing the pixel at (0,1) in chip number 12 which will result in the pixels at (17,1) being in chip number 0. The *ChipNumber* function for this organization can hence be stated as

$$ChipNumber(x,y) := (12y + x) \,\%\, 29.$$

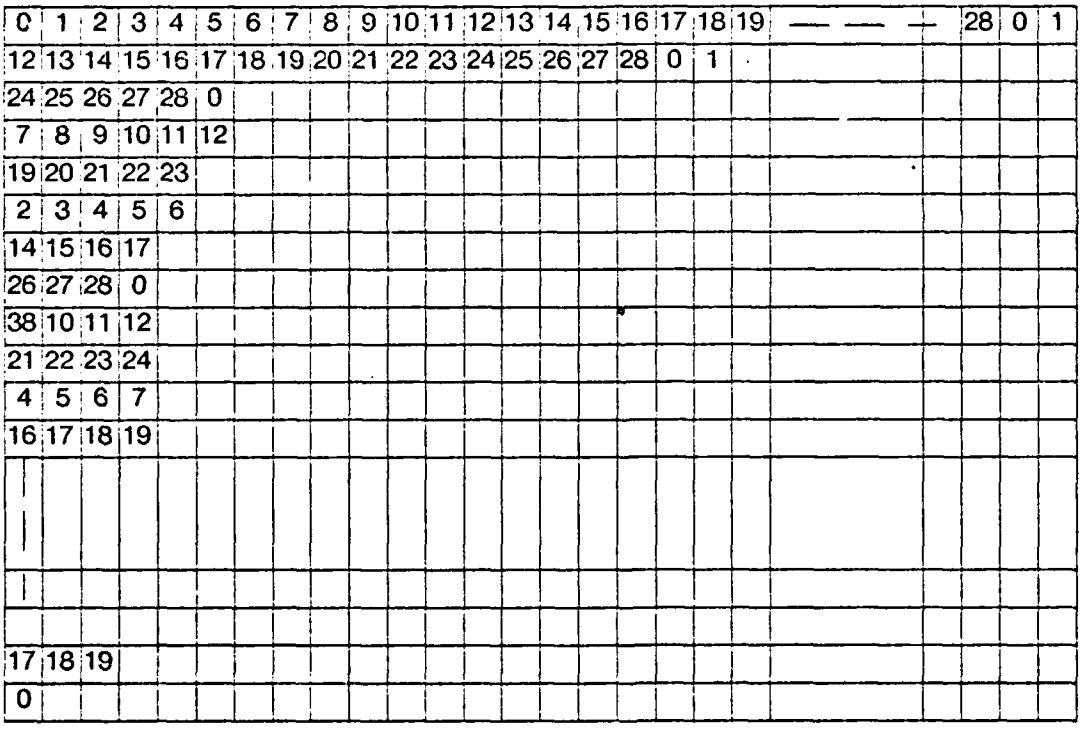


Figure 3-8: General rectangular memory organization for $N = 16$

Address and mask computations are performed in a manner similar to the Row-Column-Square organization. A pair of tables is precomputed for each rectangular shape accessed. Each pair of tables is indexed by two chip numbers and identified as *xdistdxdy* and *ydistdxdy* where *dx* x *dy* is the size of

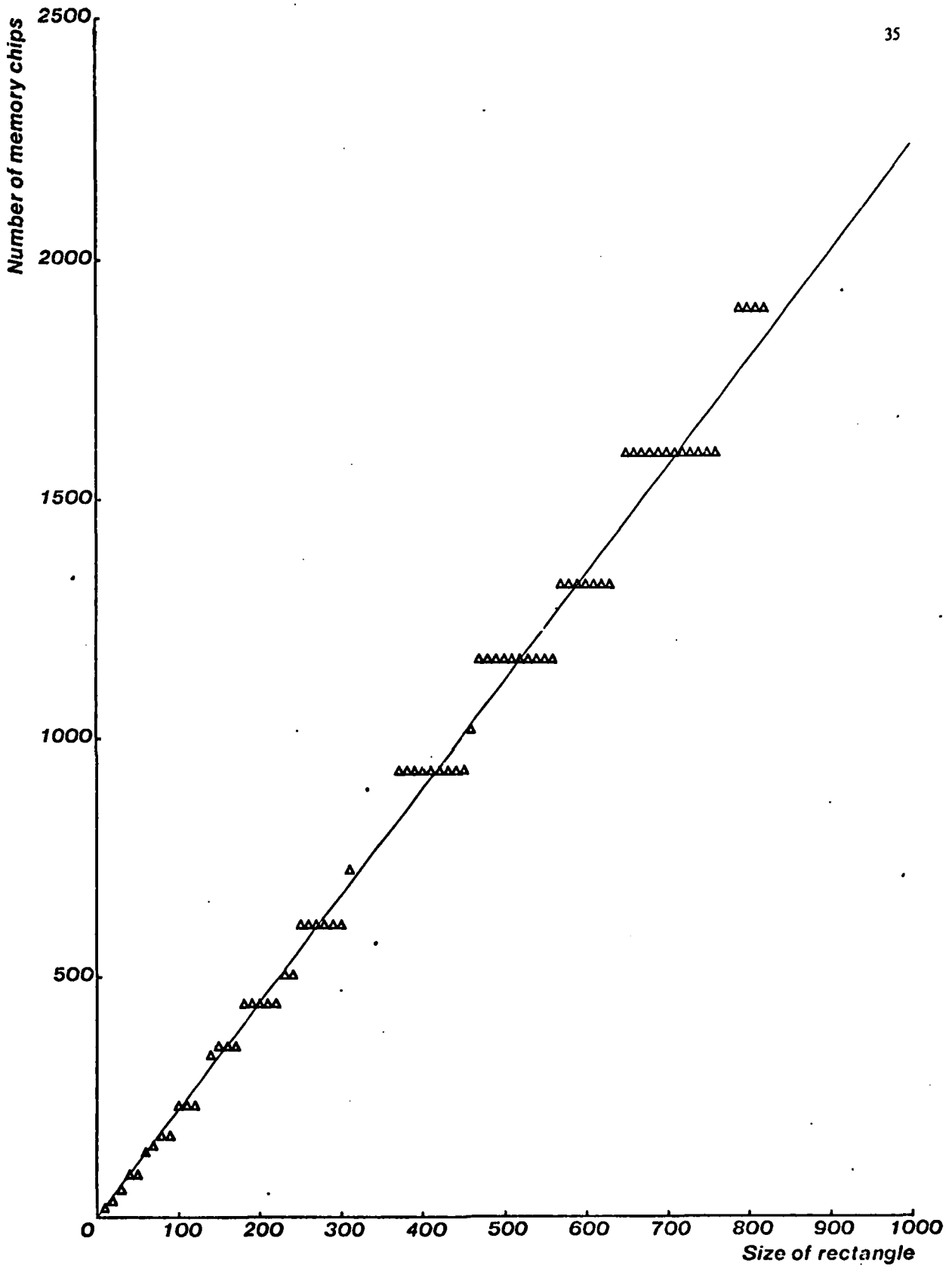


Figure 3-9: Number of memory chips for the general rectangular memory organizations

the rectangle being accessed, and the contents are similar to the contents of the *squarexdist* and *squareydist* tables of the Row-Column-Square organization. The value of *xdistdxdy*[*e*][*f*] will be the x-distance between two pixels in a *dx* x *dy* rectangle when the top corner pixel of the rectangle is in memory chip number *e* and the second pixel is in chip number *f*; *ydistdxdy*[*e*][*f*] results in the y-distance with the same constraints. Since there are approximately $2\sqrt{N}$ rectangular shapes being accessed, there will be $2\sqrt{N}$ pairs of such tables each with approximately $5N^2$ entries (assuming that $\sqrt{5}\,N$ memory chips are required to implement the memory system). Using these precomputed tables, the addressing and masking require the following functions. *C* is the number of chips actually used in the memory system.

$$d := ChipNumber(x,y)$$
$$Address(x,y,dx,dy,c) := (x + xdistdxdy[d][c]) \cdot (y + ydistdxdy[d][c])/C$$
$$Mask(x,y,dx,dy,mdx,mdy,c) := (xdistdxdy[d][c] \geq mdx) \vee (ydistdxdy[d][c] \geq mdy)$$

3.4. Conclusion

The flexible frame buffer memory organizations presented in this chapter can be used to improve the performance of displays. As can be expected, the flexibility results in an increased complexity both in the hardware implementation of the memory organization and the application software which attempts to use this flexibility. The increase in hardware complexity results from the fact that there are no easy implementations of the addressing computation, nor any obvious incremental algorithms. In Chapter 4, we will see that a large class of applications copy data read from one part of the display to another. This requires the potential movement of data from one memory chip to another. The memory organizations presented in the previous chapter have simple implementations for this data movement; they will be discussed in Chapter 4. The flexible memory organizations of this chapter result in horrible data movement patterns which need expensive implementations.

The flexible memory organizations are a high performance, high cost point in the cost-performance curve for display designs.

Chapter 4
BITBLT

A large class of display applications require extremely simple transformations on pixels already stored in the frame buffer. These transformation techniques can be used to save the often inefficient task of recomputing the image. The most common example is the task of displaying characters. If a character is already in the frame buffer, then the task of displaying the same character at a different location can be performed by merely copying the description from its previous location. The usefulness of such a mechanism increases even more when the frame buffer is larger than the displayed area and allows the storage of precomputed images that do not get displayed. All characters can now be displayed by moving individual character descriptions from fonts stored in the invisible part of the frame buffer.

The display processor associated with the frame buffer can be programmed to provide all such required transformations. The host computer chooses the transformation desired and sends its parameters to the display processor. In the case of characters, the code of the character and its destination is sufficient information for the display processor to move the character from the font definitions to the desired location.

The type of tranformations required is governed by the applications desired. We shall present a sample list of tasks and the type of transformations they require.

1. *Characters from fonts* - If fonts can be stored within the frame buffer, then placing character strings is a simply a matter of moving individual characters from the fonts to the desired locations. Standard ASCII terminals implement a similar mechanism, with the difference that they do not actually move the character descriptions but simply read the fonts while generating the video output for the CRT display. The frame buffer approach is more general because it allows the use of multiple fonts with varying sizes, scripts, and even alphabets. This application requires the ability to *copy* a rectangular region from one part of the display to another. The copy operation can also be used for other frequently used operations like clearing the display or displaying background or boundary patterns, which can be done by moving the same fixed pattern to different locations in order to cover the desired area.

2. *Scrolling, Windowing* - Scrolling images stored within frame buffers requires moving the image description in memory. The movement is necessitated by the fact that an arbitrary rectangular window might have to be scrolled and the scrolling can be in any direction [Sproull 79]. Simple scrolling operations which scroll the whole screen only in one direction can make use of hardware tricks like starting the video scan at an arbitrary scan line. Such tricks cannot be used in the general case, when the only solution is to move the contents of the frame buffer memory. Scrolling requires the rectangular *copy* operation to move part of the scrolled image to its new location (Figure 4-1). The strip emptied is now cleared by a *clear* operation or recreated by a combination of other operations. The same sequence of operations can be used to scroll in any of the four directions.

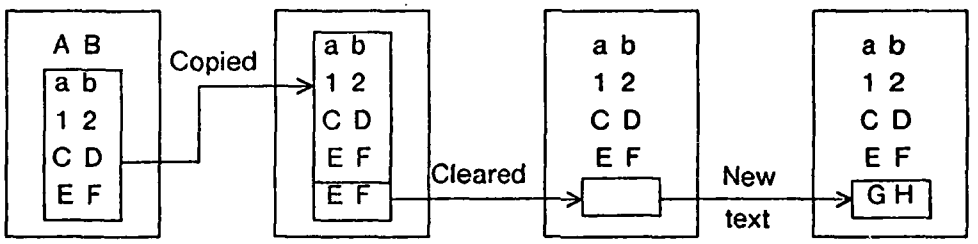


Figure 4-1: Scrolling

3. *Temporary images* - Interactive graphics makes extensive use of temporary images to convey momentary pieces of information like command menus, error messages, or status reports. These images are often put on top of other existing images which sometime also have to be displayed and almost certainly restored after the temporary image is removed. The temporary images can also be stored in the invisible portion of the frame buffer with the fonts and simply copied into the visible area when desired. A copy deletes the contents of the existing image which should be saved before performing the copy and restored when the temporary image can be removed. In order to achieve an overlay of the two images, the temporary image can either be *OR*ed or *XOR*ed into the existing image. These binary operations can be used only to combine bit-map images. The OR operation simply overlays one image on top of the other and corresponds to the logical OR when the two images have the same background color and the logical OR of one with the complement of the other in the case when images have different backgrounds. This OR operation destroys the contents of the old image which hence has to be saved and restored using the same mechanism as in the copy operation. The XOR operation preserves information because applying this operation twice will restore the contents of the original image (Figure 4-2).

4. *Cursors* - Cursors can be treated in the same manner as temporary images with the only difference being that cursors have to be moved extremely fast, usually to keep pace with some sort of pointing device. The speed requirement is the reason that most display designs provide special hardware which performs the cursor task, usually by overlaying the cursor pattern with the image during the video generation for the screen. However, if the graphics processor can perform the combination operation (including saving and restoring the original image if the information preserving transformation is not being

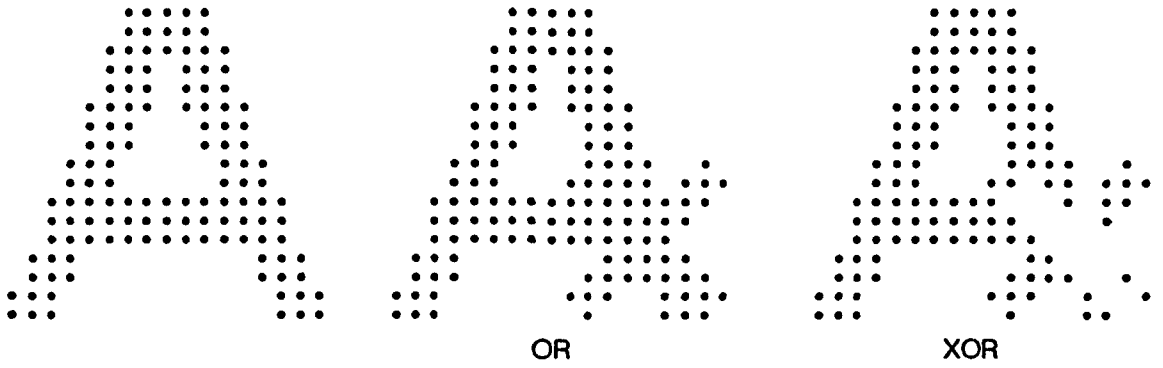


Figure 4-2: OR and XOR operations

used) fast enough, then the need for the special hardware can be eliminated. This also allows the cursor to have arbitrary shape and size, a feature not present in most special hardware implementations.

5. *Highlighting* - Instant feedback requires highlighting selected parts of the image and is usually provided by changes in intensity, background color, or by simply flashing the selected part. Intensity change can be performed in many different ways such as changing the gray-scale intensity. Flashing can be performed by alternately clearing the region and restoring the original contents. Background change requires the *invert* operation which complements the bit-map of the selected image.

6. *Browsing through large images* - Displays are often used to browse through large images which cannot be fully viewed in one screenfull. Two examples of this are viewing the checkplot of a large VLSI chip, or browsing through a large digitized terrain map. The extra space in the invisible part of the display can be used to store adjoining areas of the image that is being displayed and hence speed up small movements of the image. The standard scrolling operations can be used in the case of moving the image within the display memory. However, even the whole display memory may not be sufficient to store the entire image which has to be then stored in host computer memory or worse still in secondary memory. In this case the display and host processor have to coordinate in a fast transfer of image data to update the display memory. This operation shall be called *external transfer* and should be (if possible) as fast as the internal copy operation.

7. *Graphics* - The next few chapters of this thesis discuss the generation of graphical images through the use of precomputed primitives. Lines can be generated by joining together small strokes and filled regions by tiling together small patches. All algorithms that generate graphical images in such a manner tend to produce better output when a larger set of primitives is used although this set can be greatly reduced if these building blocks can be *mirrored* and/or *transposed* before being put into place. Once again the invisible part of the frame buffer can be used to store these primitives and they can be moved to the appropriate part of the display to form the image. The key issue in these applications is the combination function to be used when combining the stroke or the patch with the existing image. The combination function depends upon the application and is much

more complicated for gray-scale graphics than for bit-map graphics. The most popular bit-map function used is the OR operation.

4.1. BitBlt operations

All the operations discussed so far have one common theme. They take two regions in different parts of the display memory, combine them using some sort of combination function, and write the result into one of the two parts. If the two parts are identified as the source and destination, then the operation can be defined as

destination := source *op* destination

We shall restrict both source and destination areas to be rectangular and of the same size. Such an operation has been historically known as BitBlt (the name originates from the Block Transfer instruction of the PDP-10), and can be procedurally written as

BitBlt(Srcrect,Dstrect,Op),

where Srcrect and Dstrect specify the source and destination rectangular regions, and Op specifies the combination function. The Srcrect and Dstrect can overlap, in which case a valid order has to be chosen for the execution of the operation in order to preserve the correctness of the operation. As an example, if the top left corner of the Srcrect is inside the Dstrect, then a valid order for the operation is to start at the top right corner and proceed to the left and downwards. The four different possibilities are shown in Figure 4-3.

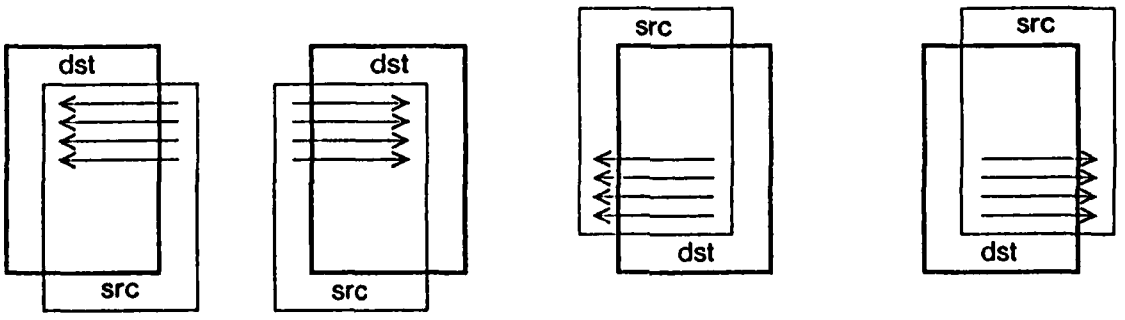


Figure 4-3: Overlapping BitBlt rectangles

The mirroring and transposition transformations provide a more general set of transformations which will then include rotation by a multiple of 90 degrees (Figure 4-4). If we allow the source

rectangle to undergo a combination of the x-mirror, y-mirror, and transposition transformations then we cannot allow the source and destination rectangles to overlap because there does not exist a valid sequence which will guarantee correct results in all circumstances.

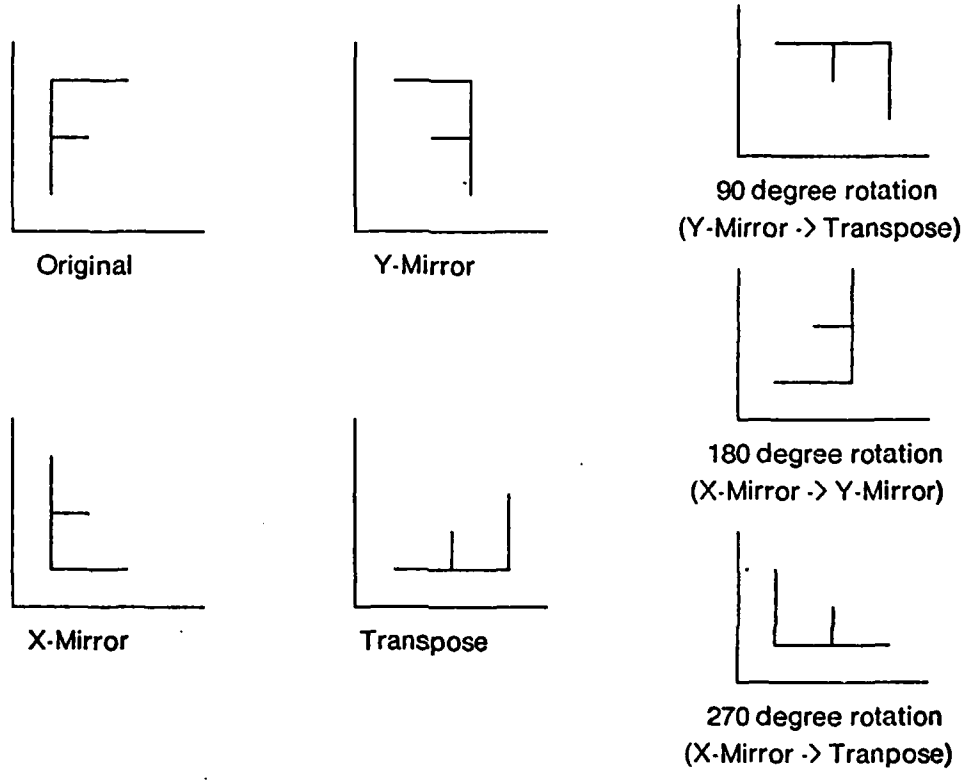


Figure 4-4: Rotation using mirroring and transposition as basis

Tables 4-1 and 4-2 tabulate some BitBlt operations and their possible applications. Table 4-1 contains boolean operations useful for bit-map displays or single-bit planes of gray-scale displays and Table 4-2 contains some of its gray-scale generalizations. The Min and Max functions are the gray-scale equivalents of the boolean AND and OR operations. If 1 is assumed to be the maximum gray-scale intensity, then (1 – intensity) is the equivalent of the logical inversion of a boolean intensity. The sum function is equivalent to the boolean XOR, because both the operations preserve the information content of the original image. Applying the XOR operation twice restores the original image, and similarly the difference operation can restore the image before the sum operation.

One possible generalization to allow more flexible geometries would be to allow trapezoidal regions instead of rectangles. This generalization will allow operations on arbitrary polygons to be

Function	Operation	Usage
Clear	Dst := constant	Clearing, Initializing
Constant	Dst := fixed patttern	Background, Boundary
Copy	Dst := Src	Text, Scrolling
Invert	Dst := NOT Dst	Highlighting
AND	Dst := Src AND Dst	A combination function
OR	Dst := Src OR Dst	Another combination function
XOR	Dst := Src XOR Dst	A non-destructive function

Table 4-1: BitBlt operations

Function	Operation	Usage
Minimum	Dst := MIN(Src,Dst)	Gray-scale AND
Maximum	Dst := MAX(Src,Dst)	Gray-scale OR
Complement	Dst := 1 − Dst	Gray-scale Invert
Sum	Dst := Src + Dst	Gray-scale XOR
Difference	Dst := Dst − Src	Gray-scale XOR
Reverse Diff	Dst := Src − Dst	Gray-scale XOR
Scale	Dst := c * Dst	change contrast
Brightness	Dst := Dst + c	change brightness

Table 4-2: Gray-Scale BitBlt operations

performed by splitting them into trapezoids by drawing horizontal lines at each vertex (Figure 4-5). The trapezoid can be further generalized by removing the restriction on the vertical segments from being straight lines to curves. This generalization allows us to perform BitBlt on geometries outlined by curves [Ball 81].

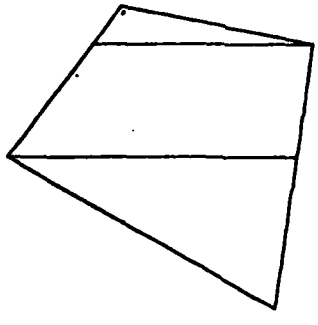


Figure 4-5: Polygons can be split up into trapezoids

4.2. BitBlt implementation

This section describes the hardware and software implementations of BitBlt. A simple loop can be used to iterate over all pixels in both the source and destination rectangular regions and perform the desired operation on each pair of pixels in the two regions. This loop should carefully choose the order in which the pixels are written in order to achieve the correct operation in the cases when the source and destination rectangles overlap. If an incorrect order is used for overlapping regions, then one of the source points could be overwritten before it is moved to its destination. The different possibilities are treated appropriately in the following procedure for BitBlt. (*xs,ys*) and (*xd,yd*) represent the top left corners of the source and destination rectangles respectively, *w* and *h* are the widths and the heights of the rectangles, and *op* is the BitBlt operation.

```
PROCEDURE BitBlt(xs,ys,xd,yd,w,h,op)
begin
    xdir := 1; ydir := 1;
    if xs < xd then begin
        xdir := -1; xs := xs + w - 1; xd := xd + w - 1;
    end;
    if ys < yd then begin
        ydir := -1; ys := ys + h - 1; yd := yd + h - 1;
    end;
    y1 := ys; y2 := yd;

    for i := 0 to h-1 do begin
        x1 := xs; x2 := xd;
        for j := 0 to w-1 do begin
            Raster(x2,y2) := Raster(x1,y1) op Raster(x2,y2)
            x1 := x1 + xdir; x2 := x2 + xdir;
        end;
        y1 := y1 + ydir; y2 := y2 + ydir;
    end;
end;
```

The mirroring and transposition can be easily accommodated into this procedure because mirroring only changes the direction of one of the two nested loops and transposition transposes the loops for either the source or the destination rectangle.

All BitBlt operations can be performed on many pixels in parallel provided the display architecture allows parallel updates. Such parallelism can greatly enhance the performance and is even necessary for some interactive applications. Chapter 2 discussed several parallel architectures which allow the update of several pixels in each memory cycle. Such architectures are implemented by splitting the frame buffer into several memory chips; each memory chip can be read or written during each memory cycle.

Parallel implementations introduce two new problems. The first one is that a pixel read from one memory chip might have to be combined with a pixel in a different memory segment. This is due to the fact that the source and destination rectangles may not have the same alignment with respect to the word boundaries imposed by the parallel architecture. The second is the need for the ability to update a subset of the memory chips in order to accommodate small area BitBlts and the end-conditions of larger ones.

The next four sub-sections discuss the parallel implementations for the *scan-line* and the *square* memory organizations. For each of these organizations, the first algorithm presented assumes the ability to make arbitrarily aligned memory accesses; the second algorithm restricts the access to word boundaries.

4.2.1. Nx1 arbitrary access

This memory organizations allows us to access an arbitrary set of N pixels along a scan-line of the display. This would modify the inner loop of BitBlt to jump by N pixels instead of one. We shall use the term RasterWord(x,y) to refer to a piece of word accessed with its left corner located at (x,y). The operation *Align* aligns the pixels read at ($x1$,$y1$) for writing at ($x2$,$y2$). This operation requires the movement of data between memory chips and shall be the topic of discussion in Section 4.3.

```
PROCEDURE BitBlt(xs,ys,xd,yd,w,h,op)
{Nx1 arbitrary access}
begin
    xdir := N; ydir := 1;
    if xs < xd then begin
        xdir := -N; xs := xs + w - N; xd := xd + w - N;
    end;
    if ys < yd then begin
        ydir := -1; ys := ys + h - 1; yd := yd + h - 1;
    end;
    y1 := ys; y2 := yd;

    for i := 0 to h-1 do begin
        x1 := xs; x2 := xd;
        for j := 0 to w/N do begin
            RasterWord(x2,y2) := Align(RasterWord(x1,y1))
                                          op RasterWord(x2,y2)
            x1 := x1 + xdir; x2 := x2 + xdir;
        end;
        y1 := y1 + ydir; y2 := y2 + ydir;
    end;
end;
```

The procedure in the above exposition updates only words and would hence result in an error in

the last iteration of the loop if the width w is not an exact multiple of N. The last iteration should update $w\%N$ points; these points are left justified if xinc is 1 and right justified if xinc is -1. An N-bit variable *mask* is used to allow the update of partial words. The presence of a 0-bit in the appropriate position is required to update that bit of the display memory word.

```
PROCEDURE BitBlt(xs,ys,xd,yd,w,h,op)
{Nx1 arbitrary access with masking}
begin
    xdir := N; ydir := N;
    endmask := (2^N-1) rshift w%N;
    if xs < xd then begin
        xdir := -N; xs := xs + w - N; xd := xd + w - N;
        endmask := (2^N-1) lshift w%N;
    end;
    if ys < yd then begin
        ydir := -1; ys := ys + h - 1; yd := yd + h - 1;
    end;
    y1 := ys; y2 := yd;

    for i := 0 to h-1 do begin
        mask := 2^N-1;
        x1 := xs; x2 := xd;
        for j := 0 to w/N do begin
            if (j = w/N) then mask = endmask;
            RasterWord(x2,y2) := Align(RasterWord(x1,y1))
                                        op RasterWord(x2,y2)
            x1 := x1 + xdir; x2 := x2 + xdir;
        end;
        y1 := y1 + ydir; y2 := y2 + ydir;
    end;
end;
```

The *Align* operation for the Nx1 organization requires a circular shift of the word read (Figure 4-6). The shift count is the difference in alignment of the two rectangles with respect to the word boundary.

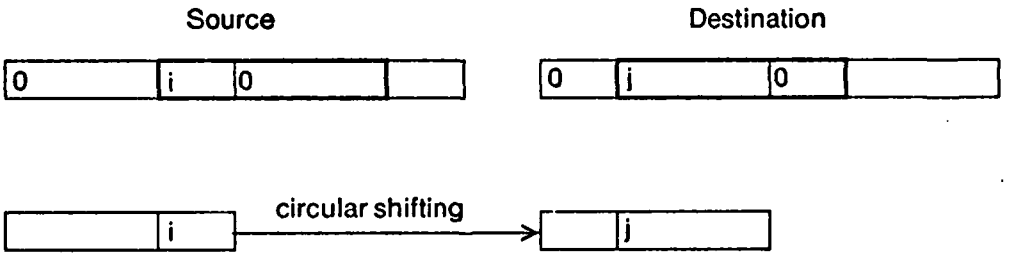


Figure 4-6: Aligning Nx1 rectangles

Of the mirroring and transposition transformations, the x-mirror merely changes the direction of the source y-loop, and the y-mirror changes the direction of the x-loop and also requires the source word to be mirrored in addition to being aligned with respect to the destination word. The transposition transformation cannot make use of the parallelism offered by this memory organization because this memory organization does not allow a word read along the x-direction to be written along the y-direction.

4.2.2. Nx1 fixed access

In this memory organization the N pixels accessable in one memory access are forced to lie between Nx1 boundaries of the screen (Figure 2-1). If the source and destination rectangles are misaligned with respect to the fixed boundaries, then a destination word that can be written in one memory cycle cannot be read from the source in one memory cycle. Similarly a source word that can be read in one memory cycle cannot be written in one memory cycle into its destination. Because some operations choose to read both source and destination and write a combined value into the destination, we choose to read two adjacent source words and combine them for each destination word (Figure 4-7). Notice that when we attempt to write the adjacent destination word we have to once again read one of the source words read for the previous destination word. This can be used to speed up the inner loop by saving one of the source words and only reading one source word for each iteration in the inner loop. This organization can hence provide the same bandwidth as the organization which allows arbitrary Nx1 accesses at essentially no extra cost.

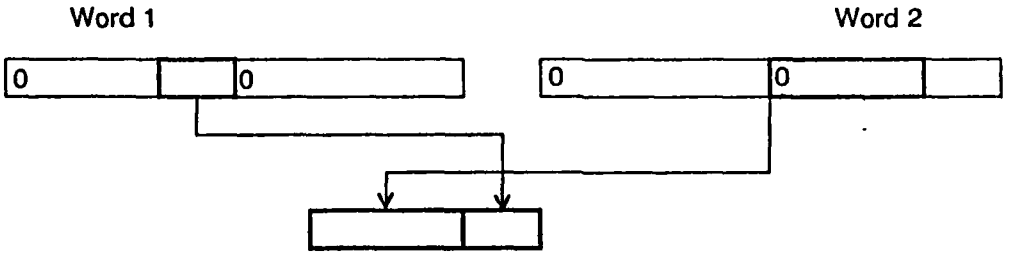


Figure 4-7: Each Nx1 destination uses two source words in the fixed access organization

4.2.3. MxM arbitrary access

Small area BitBlts (characters for example) prefer a square memory organization because updates in such an organization can be performed in fewer memory cycles. The *M*x*M* arbitrary access memory organization allows the access and update of an arbitrary *M*x*M* square on the display. The BitBlt procedure is very similar to the BitBlt procedure in the *N*x1 arbitrary access organization. Two masks *xmask* and *ymask* are ORed to produce the *M*x*M* mask which is used as the write-enables for the memory chips (Figure 4-8). The operation *Align* aligns the pixels read (*x*1,*y*1) for writing at (*x*2,*y*2).

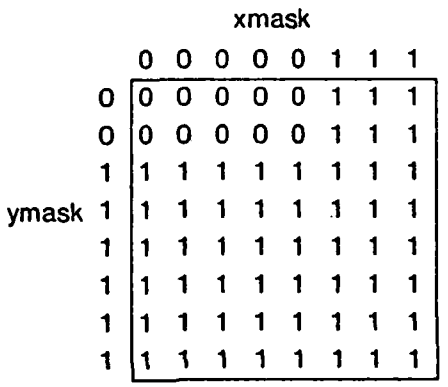


Figure 4-8: Masking *M*x*M* squares

```
PROCEDURE BitBlt(xs,ys,xd,yd,w,h,op)
{MxM arbitrary access with masking}
begin
    xdir := M; ydir := M;
    endxmask := (2^M-1) rshift (w%M);
    endymask := (2^M-1) rshift (h%M);
    if (xs < xd) then begin
        xdir := -M; xs := xs + w - M; xd := xd + w - M;
        endxmask := (2^M-1) lshift (w%M);
    end;
    if (ys < yd) then begin
        ydir := -M; ys := ys + h - M; yd := yd + h - M;
        endymask := (2^M-1) lshift (h%M);
    end;

    y1 := ys; y2 := yd;
    ymask = 2^M-1;
    for i := 0 to h/M do begin
        if (i = h/M) then ymask := endymask;
        xmask := 2^M-1;
        x1 := xs; x2 := xd;
        for i := 0 to w/M do begin
            if (j = w/M) then xmask := endxmask;
            RasterWord(x2,y2) := Align(RasterWord(x1,y1))
                                        op RasterWord(x2,y2);
            x1 := x1 + xdir; x2 := x2 + xdir;
        end;
        y1 := y1 + ydir; y2 := y2 + ydir;
    end;
end;
```

This BitBlt procedure comprises two nested loops each of which jumps by M pixels. The inner loop moves along the x-direction while the outer loop moves along the y-direction. The fast movement along the x-direction is necessitated by the fact that most of the time the image being scrolled contains text, which is expected to change from top to bottom and not from left to right. This loop structure provides the expected effect. If BitBlt were fast enough that the direction of movement would be unnoticeable, we would then have to worry about the CRT video scanning interacting with the BitBlt memory updating. So if we were actually scrolling the whole display's contents and the inner loop incremented in the y-direction, then the lower half of the display may appear changed while the upper half may still be unchanged. Choosing the inner loop to move in the x-direction and properly synchronizing to the video scan can in fact make the scrolling appear to be instantaneous.

As in the case of the Nx1 organization, the x-mirror and y-mirror transformations change the directions of the loops and require the ability to mirror MxM words, although as we have discussed before the source and destination rectangles cannot overlap. The transposition transformation can be done by having the source rectangle loop in one direction and the destination rectangle loop in the other direction and having the ability to transpose single MxM words.

The alignment operation in the square organization requires a two-dimensional circular shift (Figure 4-9). The shift counts in each dimension are the alignment differences between the source and destination rectangles along those dimensions. The pixels marked in the figure by a small circle, small square, and a large square are read from memory chips marked by the same symbols, and written into a different set of memory chips. The two-dimensional circular shift aligns these pixels from the memory chips in the source rectangle to the ones in the destination rectangle.

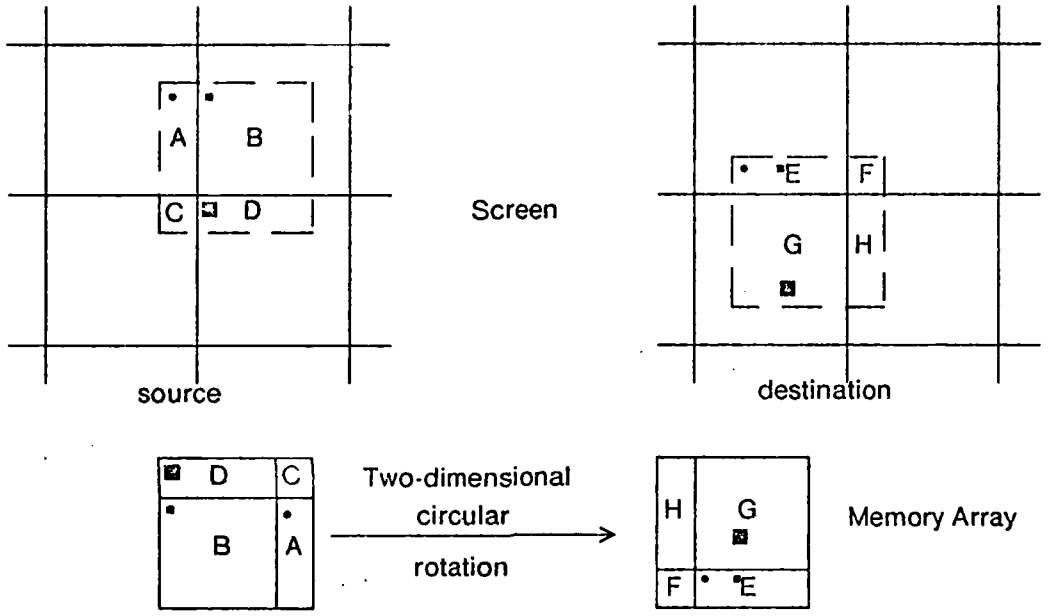


Figure 4-9: Aligning MxM squares

4.2.4. MxM fixed access

When the square organization is restricted to access words that have to be aligned to MxM word boundaries, then the inner loop of BitBlt has to combine four adjacent source words to update one destination word. The optimization technique used for the Nx1 fixed access organization is still applicable because when we attempt to write adjacent words, three out of four source required would have probably already been read to update previous destination words. However, saving these could be a problem because a destination word needs the current source word to be combined with source words both from the previous column and the previous row. The word from the previous column was read in the previous memory cycle and is hence readily available even if we save only one word of memory. The words in the previous row could have been read a full x-scan before and saving them

could potentially involve storing M complete scan-lines. If we discard this possibility, but save the word read in the previous memory access then we lose 1/4 the memory bandwidth on the average. We can still use the entire memory bandwidth when the source and destination rectangles are exactly word aligned in the y-direction (during a horizontal scroll for example). But if the two rectangles are misaligned by exactly $M/2$, then only 1/2 the bandwidth can be used.

4.3. BitBlt communication

The transformation operations required by all BitBlt algorithms require the movement of data between memory chips in order to align reads and writes. This section shall discuss the requirements for the *scan-line* and *square* memory organizations.

4.3.1. Nx1 communication

The Nx1 memory organization requires N memory chips which can be used in parallel to read or write N pixels which lie along a scan-line of the display. Depending upon the addressing scheme used these N pixels can be constrained to lie within a fixed Nx1 grid on the display or be free to originate at any arbitrary pixel. In both these variations of the Nx1 memory organiztion, a circular shift is required to align an Nx1 word so that it can be updated at a different location. The shift count is the alignment difference between the two words. The y-mirror operation also requires the ability to mirror the words.

There are several alternatives for implementing these transformations which vary in speed and space required in terms of logic and pins. There are two classes of implementation alternatives. The first class of alternatives performs the tranformations externally after reading the word to be rotated from the memory and then writing the rotated word back. The second manipulates the data within the memory chips by connecting them suitably to each other.

The obvious external implementation takes the N pixels from the N memory chips, performs the shifting/mirroring externally and returns the N transformed pixels to the memory (Figure 4-10). The external rotation can be performed using a barrel shifter or one of the other well known shifting schemes. The mirroring merely adds another level of multiplexing.

The most obvious internal implementation connects the memory chips to both their adjacent neighbors and performs the shifting by sequentially shifting the data from chip to chip (Figure 4-11).

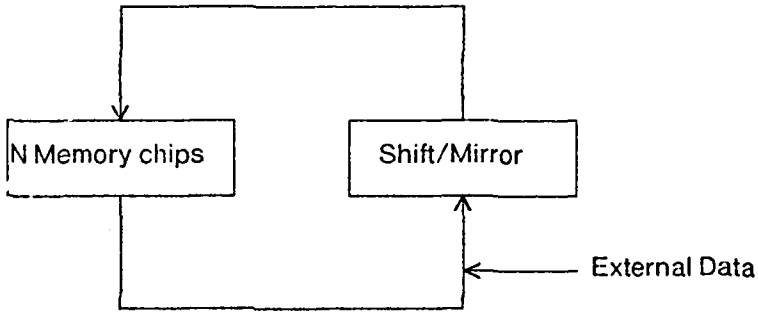


Figure 4-10: External transformation in the Nx1 organization

This simple scheme requires $2g$ pins in each memory chip (g is the number of bits/pixel) and takes a maximum of $N/2$ steps to perform. The average number of steps required is $N/4$. Adding more pins can improve this performance (Section 4.3.3).

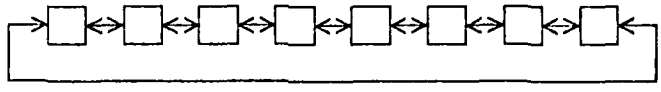


Figure 4-11: Neighbour connections in the Nx1 organization

4.3.2. MxM communication

The MxM memory organization is implemented using M^2 memory chips and provides the ability to read MxM squares of the display. The memory can be addressed to provide only word-aligned squares or arbitrary squares with the additional variation of staggering the squares to also provide horizontal spans. The square organization requires a two-dimensional circular shift to align an MxM word with a different word (the situation for the staggered addressing is slightly different in which case a y-directional shift of selected columns is also necessary). The rotation counts in each dimension depend upon the alignment differences in those dimensions. In addition we also need the x-mirror, y-mirror, and transposition tranformations to implement the mirroring and transposition.

There exist several possibilities for implementing the transformations external to the memory chip array. The shifting can be performed sequentially in M-bit segments which would take M time steps. A parallel implementation using $2M$ shifters, M for one dimension and M for the other, can

implement the shifting in one time step. A slightly different implementation takes two time steps by time-multiplexing between one set of M shifters, but the cost of the multiplexing makes this scheme as expensive as the previous one. Any one of these shifting schemes has to be followed by multiplexing to implement the mirroring and transposition transformations.

An attractive connection mechanism for performing this transformation uses only two sets of M wires to connect the M^2 memory chips (Figure 4-12). The dark set of wires can be used to read out any row or column, and the dotted wires can be used to write into any row or column. The external network can shift/mirror the row or column read. This technique requires only $2g$ wires in each memory chip but takes M time steps. Only four control wires per memory chip are required to control this mechanism. Two of these wires are bussed along rows and the other two are bussed along columns. One of each selects whether the chip should enable its data onto the dark wire and the second selects whether the chip should latch the contents of the dotted wire.

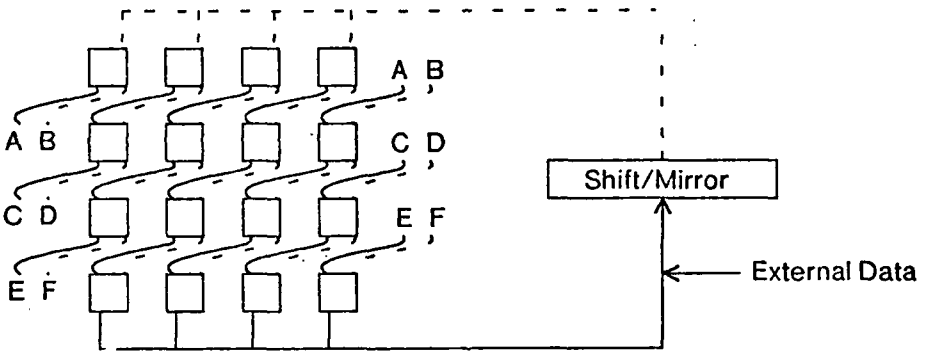


Figure 4-12: Sequential transformation in the MxM organization

A simpler connection mechanism connects each memory to its four immediate neighbours (Figure 4-13). The mirroring/transposition requires appropriate multiplexing at the edges. This technique requires $4g$ data wires per memory chip and only two control wires which encode the direction of transfer. This mechanism takes a maximum of M and an average of $M/2$ steps to perform the two dimensional shifting required to align the data. Mirroring and transposition always require M time steps.

The neighbor connection mechanism can be made more efficient by using tri-state wires and using a scheme shown in Figure 4-14. This improvement allows each chip to communicate with any of its eight neighbors. The two dimensional shifting now only requires a maximum of $M/2$ and an average

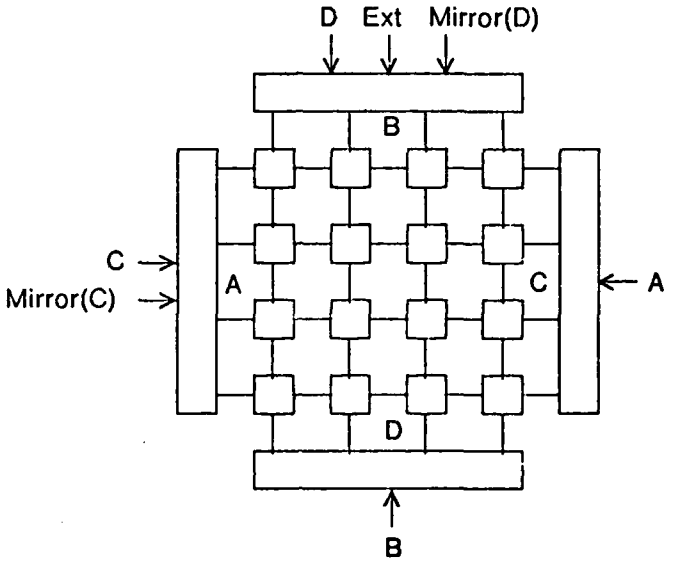


Figure 4-13: Neigbour connections in the MxM organization

of $M/4$ steps. Mirroring and transposition still require M steps. The control now needs three wires to encode the eight possible directions of data transfer.

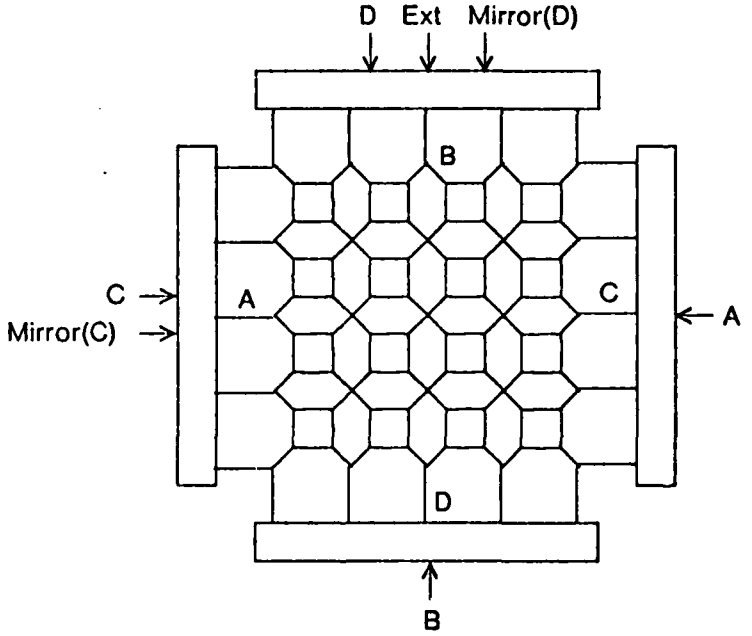


Figure 4-14: Tri-state neighbour connections in the MxM organization

All the above connections need another wire for enabling or disabling the transfer when the staggered addressing is being used in order to remove the stagger after accessing an MxM square.

4.3.3. Non-neighbor communication

Non-neighbor communication can be used to achieve higher performance for the circular shifting operation. Section 4.3.1 showed that in the nearest neighbor communication paths of the Nx1 memory organization, a shift takes a maximum of $N/2$ and an average of $N/4$ steps. If we increase the number of connections to four neighbors, then the question is what neighbors to choose. Symmetry arguments dictate that the connections on one side will be the same as the ones on the other. One choice of connections is to choose the immediate neighbors and the ones $\sqrt{N}$ away. Using this interconnection scheme, an arbitrary location can be reached in a maximum of $\sqrt{N}$ moves and an average of $\sqrt{N}/2$ moves.

We can, however, always improve on the $\sqrt{N}/2$ bound for the average number of steps by a constant factor. Assume that the neighbors are at a distance of a and b in both directions. In one shift step, the data can be moved from point 0 to $-a$, $-b$, a, and b. In two steps,the data can reach $-2a$, $-a-b$, $-a+b$, $-2b$, $-b+a$, $2a$, $a+b$, and $2b$. In general the ith step reaches $4i$ points. If all these points were indeed disjoint,then k steps would span a total of

$$2k^2 + 2k + 1$$

points. If this total was equal to N, then k is asymptotically equal to $\sqrt{N/2}$, and the average number of steps to reach any point is $\sqrt{2N}/3$. This bound is only slightly better than $\sqrt{N}/2$, but can almost always be achieved. Table 4-3 shows the values of a and b which result in the minimum number of moves for different values of N. It also tabulates the average number of moves Av, and the ratio $Av/\sqrt{N}$, which asymptotically approaches $\sqrt{2}/3$.

These results can be extended to more neighbors and also to two-dimensional neighbors. In the extreme case each memory chip can be connected all other memory chips, in which case all shifts take only one step.

N	a	b	Av	$Av/\sqrt{N}$
8	1	2	1.250000	0.441942
16	1	6	1.812500	0.453125
24	3	4	2.208333	0.450774
32	1	7	2.625000	0.464039
40	4	5	2.900000	0.458530
48	1	20	3.229167	0.466090
56	1	10	3.482143	0.465321
64	1	14	3.718750	0.464844
72	1	11	3.972222	0.468131
80	1	22	4.175000	0.466779
88	1	26	4.375000	0.466377
96	1	42	4.593750	0.468848
104	1	16	4.778846	0.468604
112	7	8	4.937500	0.466550
120	1	14	5.133333	0.468607
128	1	15	5.312500	0.469563

Table 4-3: Average moves for four neighbor connections.

4.3.4. Overlapping communication with memory access

The inner loop of all the algorithms for BitBlt requires an *Align* operation between a pair of read and write memory cycles. In the presentation of the algorithms, the read, align, and write operations are performed sequentially. A slow alignment mechanism hence becomes a bottleneck of BitBlt performance. For BitBlt areas much larger than the area accessed in a memory cycle, the three operations are performed repeatedly. An optimization of the inner loop is to overlap a read and write operation with an align operation. A sequence of

Read, Align, Write, Read, Align, Write, Read, Align, Write, Read, Align, Write

is now transformed into

Read, (Align/Read), (Write/Align), Read, (Write/Align), Read, (Write/Align), Write

If an align operation takes longer than a write followed by a read memory cycle, then the memory is idle for some of the time. Conversely, the align operation does not have to be shorter than a write and read memory cycle.

In the case of the MxM organization for $M=8$, the tri-state neighbor connection mechanism takes a maximum of 4 steps for alignment. This is approximately the same as the time required for two memory cycles which take 2 steps each because of the multiplexed addressing.

4.4. BitBlt performance

This section discusses the performance of BitBlt for the various memory organizations described in this chapter. The memory organizations can be compared only for the same cost measure which can be considered to be the number of chips in the memory system. The Nx1 organization uses N memory chips and the MxM organization uses M^2 memory chips. We shall base all comparison for two different cost measures, one for 16 memory chips and for 64 memory chips. N has a value of 16 and 64 in the two cost measures for the Nx1 organization while M has a value of 4 and 8 for the MxM organization.

We shall use two criteria for comparison, the first criterion based on the performance of small area BitBlts (e.g. 8x10 characters), and the second based on large area ones (e.g., scrolling a full 1024x1024 display). If the alignment operation is overlapped with the memory cycles, then the performance is directly related to the percentage of the memory bandwidth that can be used for updates.

4.4.1. Small area BitBlt

An 8x10 character will be used to illustrate the performance of small area BitBlts in the various memory organizations. We shall first discuss the memory systems using 16 memory chips and then discuss the larger memory system with 64 memory chips.

In a 16 chip memory system:

- with a 16x1 *fixed access* memory organization, updating an 8x10 character will take 10 or 20 memory cycles depending on whether the character crosses the word boundary or not. The probability of the character crossing the boundary is about 7/16, and hence the average number of memory cycles required is 14.375.

- with a 16x1 *arbitrary access* memory organization, updating an 8x10 character will always take 10 memory cycles.

- with a the 4x4 *fixed access* memory organization, updating an 8x10 character will take a minimum of 6 memory cycles when the word boundaries match with the character and a maximum of 12 memory cycles in the pathetic case of all boundaries mismatching. Assuming that all pixel locations are equally likely, the average number of memory cycles is 8.9.

- with a the 4x4 *arbitrary access* memory organization, updating an 8x10 character always takes 6 memory cycles.

In a 64 chip memory system

- with a 64x1 *fixed access* memory organization, updating an 8x10 character will probably take 10 memory cycles but it might take 20 in the unlikely case of the character crossing the word boundary. Taking probabilities into account the character can be written in an average of 11.1 memory cycles.

- with a 64x1 *arbitrary access* memory organization, updating an 8x10 character will always take 10 memory cycle.

- with a 8x8 *fixed access* memory organization, updating an 8x10 character takes a minimum of 2 memory cycles, a maximum of 6 memory cycles, and an average of 4.1 memory cycles.

- with a 8x8 *arbitrary access* memory organization, updating an 8x10 character will always take 2 memory cycles.

These results are summarized in Table 4-4.

Memory chips	16	64
*N*x1 fixed access	14.4	11.1
*N*x1 arbitrary access	10	10
*M*x*M* fixed access	8.9	4.1
*M*x*M* arbitrary access	6	2

Table 4-4: Average number of memory cycles to update an 8x10 character, for different memory organizations with 16 and 64 memory chips.

4.4.2. Large area BitBlt

Scrolling a large display region will be analyzed to compare the performance of various memory organizations for large-area BitBlts. Large-area BitBlts repeat the inner loop a large number of times and the end-conditions of the area do not significantly affect the performance. The performance can be essentially compared based on the utilization of the memory accesses in the inner loop.

The inner loops of BitBlt update

- *N* pixels in the *N*x1 *fixed access* memory organization. The BitBlt procedure has to save *N* pixels from consecutive memory accesses in order to maintain the 100% utilization.

- *N* pixels in the *N*x1 *arbitrary access* memory organization. No buffering is necessary. The memory organization hence utilizes 100% of the memory bandwidth.

- an average of 3/4 out of the *M*x*M* pixels in the *M*x*M fixed access* memory organization when we buffer only one *M*x*M* word. The memory organization only utilizes 75% of the memory bandwidth.

- all of the MxM pixels in the MxM *arbitrary access* memory organization and no buffering is required. This memory organization also utilizes 100% of the memory bandwidth.

4.4.3. Analysis

From the discussion of large area BitBlts, it is obvious that if we choose the MxM organization then we should organize the addressing to access arbitrarily aligned squares, and if we were to choose the Nx1 organization we could use either the fixed or the arbitrary access addressing. From the discussion of small area BitBlts, we can conclude that for the 16 chip system all memory organizations are approximately equally good within a factor of 2, and for the 64 chip system the square organization is preferable to the linear organization by almost a factor of 5 for characters.

The overall conclusion is that if we were designing a cheap low bandwidth memory system, we would choose the linear fixed access memory organization. If we were designing a high bandwidth system we would use the square arbitrary access memory organization. These design decisions are indeed true for the SUN work-station [Bechtolsheim 80] which uses the 16x1 fixed access memory organization, and the 8x8 display [Sutherland 81] which uses the square arbitary access memory organization.

Chapter 5
Line Drawing

Line drawing is perhaps the most frequent of the graphical operations used to create illustrations on computer displays. It is the only operation provided by calligraphic displays; all images displayed on such displays have to be created as line drawings. On raster displays, lines are approximated by intensifying an appropriate set of pixels. The desire to achieve extremely high line-drawing speeds motivated this study of a variety of different algorithms that can be used to draw lines. These algorithms can also be used for other scan-conversion tasks such as the filling of polygons and the drawing of parametric curves.

Displaying a set of pixels by merely turning them *on* is the most common form of displaying lines. Such lines are called *bit-map lines*, and are drawn by displaying the pixels that lie close to the desired line. Such lines show "jaggies" when the lines jump between rows or columns of pixels. This jagginess is an artifact of the sampling error that occurs when a signal is not sampled at the appropriate resolution, and is known as *aliasing* [Crow 76]. Bit-map lines can be smoothed by using gray-scale pixels and intensifying these pixels at different intensities to produce aesthetically improved lines. The use of intermediately intensified pixels can hence be used to effectively smooth the row or column jumps observed in bit-map lines. Such lines are called *gray-scale* or *anti-aliased* lines. This chapter discusses bit-map lines, while anti-aliasing is the topic of the next two chapters.

To achieve very high line-drawing speeds, we need the ability to update several pixels in parallel. The various memory organizations we discussed earlier provide this ability. We now need algorithms which can allow us to use this ability. Conventional line-drawing algorithms draw out the line by sequentially determining the location of the next pixel to be intensified. This chapter will discuss various algorithms that generate more than one pixel at a time, and their applicability under the various memory organizations.

We shall restrict all line-drawing discussions to lines drawn from the coordinates (0,0) to (dx,dy). This assumption causes no loss of generality because lines with different origins are translations of

lines with the origin (0,0). We shall also assume that the lines drawn lie in the first octant, i.e., $0 \leq dy \leq dx$. This assumption also loses no generality, because lines in the other octants can be generated using a combination of reflection and transposition transformations.

5.1. Bit-map lines

A line in the first octant illuminates one pixel in each vertical column. This assumption ensures a line of nearly constant width and brightness. So the line drawing problem is to compute the value of y_i for each x_i between 0 and dx. The optimal point to be illuminated is the one which lies closest to the line. The closest point is the one that has the smallest perpendicular distance to the line. However, for any given line the vertical distance is related to the perpendicular distance by a constant factor ($= dx/\text{sqrt}(dx^2+dy^2)$). Consequently, minimizing the vertical distance to the line is sufficient to produce the optimal line. The vertical distance is minimized by rounding the actual y-value of the line at x_i to produce the value of y_i. This leads to the following obvious algorithm which enumerates the optimal line.

```
for x_i := 0 to dx do begin
    y_i := round( (dy/dx) * x_i);
    display(x_i,y_i);
    end
```

The maximum vertical error for this algorithm is hence 1/2.

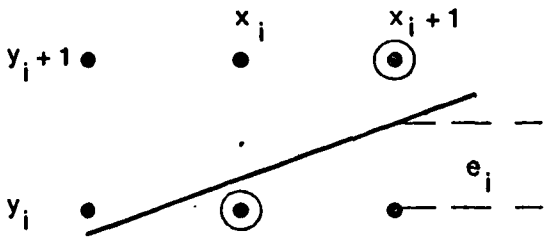


Figure 5-1: Incremental line drawing.

Simple program transformations can be used to transform this algorithm to the well known line drawing algorithms [Sproull 81]. The first transformation of this series changes the multiplication into an incremental addition such that the value of y_{i+1} is computed by incrementing the value of y_i. For lines in the first octant the point in the next column is either horizontally or diagonally across from the current column (see Figure 5-1). The decision of whether it is horizontally or diagonally across depends upon the value of e_i which is the vertical distance between the value of the line at

x_{i+1} and y_i. If this value is greater than or equal to 1/2 then the next point is diagonally across, else it is horizontally across. The value of e_i is updated appropriately depending upon the direction of the move. The following algorithm reflects this change to an incremental algorithm.

```
y_i := 0;
e_i := (dy/dx);                    { initial error at x_i }
for x_i := 0 to dx do begin
    display(x_i,y_i);
    if (e_i ≥ 1/2) then begin
        y_i := y_i + 1;
        e_i := e_i + (dy/dx) - 1;
        end
    else begin
        e_i := e_i + (dy/dx);
        end
    end
```

Substituting $r = 2(e_i - 1/2)dx$ converts this algorithm into the well known Bresenham algorithm [Bresenham 65].

```
y_i := 0;
r := 2*dy - dx;
for x_i := 0 to dx do begin
    display(x_i,y_i);
    if (r ≥ 0) then begin
        y_i := y_i + 1;
        r := r + (2*dy - 2*dx);
        end
    else begin
        r := r + (2*dy);
        end
    end
```

The Bresenham algorithm is an integer algorithm which involves only one addition and comparison in the inner loop (the expressions in parentheses can be precomputed). It generates one point of the optimal line in each iteration of the inner loop. The variable r is a scaled measure of the vertical error, the distance from the line to the pixel illuminated.

5.2. Measure of appearance

Bit-map lines show annoying jaggedness when they jump across rows or columns. Lines which are close to being horizontal or vertical are much more annoying then the lines that are either exactly horizontal or vertical or close to being diagonal. This section identifies some measures that can be used to characterize the appearance of bit-map lines.

Before attempting to characterize the appearance of the lines produced by any given algorithm, a

minimum set of acceptance criteria have to satisfied by all the lines produced by the algorithm. The most important of these requirements is the matching of the two end-points, which means that the two points between which the line is supposed to be drawn should definitely be displayed. Two other requirements are that the line should be monotone and should not have any gaps or jumps. For the line from (0,0) to (dx,dy) in the first octant ($0 \leq dy \leq dx$), these requirements impose the following constraints

1. $y_0 = 0$
2. $y_{dx} = dy$
3. Forall i suchthat $1 \leq i \leq dx$
   $0 \leq (y_i - y_{i-1}) \leq 1$

The Bresenham algorithm meets all these requirements.

The algorithms typically used to draw lines are designed to minimize the error from the ideal line. The Bresenham algorithm, for example, ensures that the maximum deviation from a line to any pixel displayed is less than or equal to 1/2. The maximum deviation from the ideal line hence provides a measure of the appearance of the lines produced. This measure is fairly good to the first order because the same line drawing algorithms produce better-looking lines on displays with higher resolutions, where the actual screen distance of the maximum deviation is the only parameter getting reduced. The average deviation from the ideal line can be used as a measure of the distribution of the error. Other statistical norms can also be used to measure this distribution.

Much of the unpleasant appearance of lines is due to the wavy appearance of the jaggies, which is not captured by any of the measures we have discussed so far. One measure that captures the wavy nature of signals is the Fourier Transform. The Fourier Transform of a wavy signal shows a peak at the fundamental frequency of the wave. Figure 5-2 shows three different lines, the distribution of the absolute vertical error over the lines, and the Fourier Transform of the error distributions. Notice that the shallow line shows a low frequency wave, and the steep line shows a high frequency wavy behaviour. The *zero frequency* (DC) component of the Fourier Transform is the average error from the ideal line.

The sum of the Fourier Transforms for all lines with different slopes (i.e., for a given dx, lines such that $0 \leq dy \leq dx$) adds up to a constant because of the shift towards higher frequencies as the line gets steeper. The sum is plotted in Figure 5-3. The zero frequency component of the sum (not shown in the graph but indicated in the figure), indicates the average error for lines with all slopes. This sum of the Fourier Transforms is a measure that will be used to compare different line drawing algorithms, because it can used to find out if the algorithm is adding an undesirable ripple onto the line.

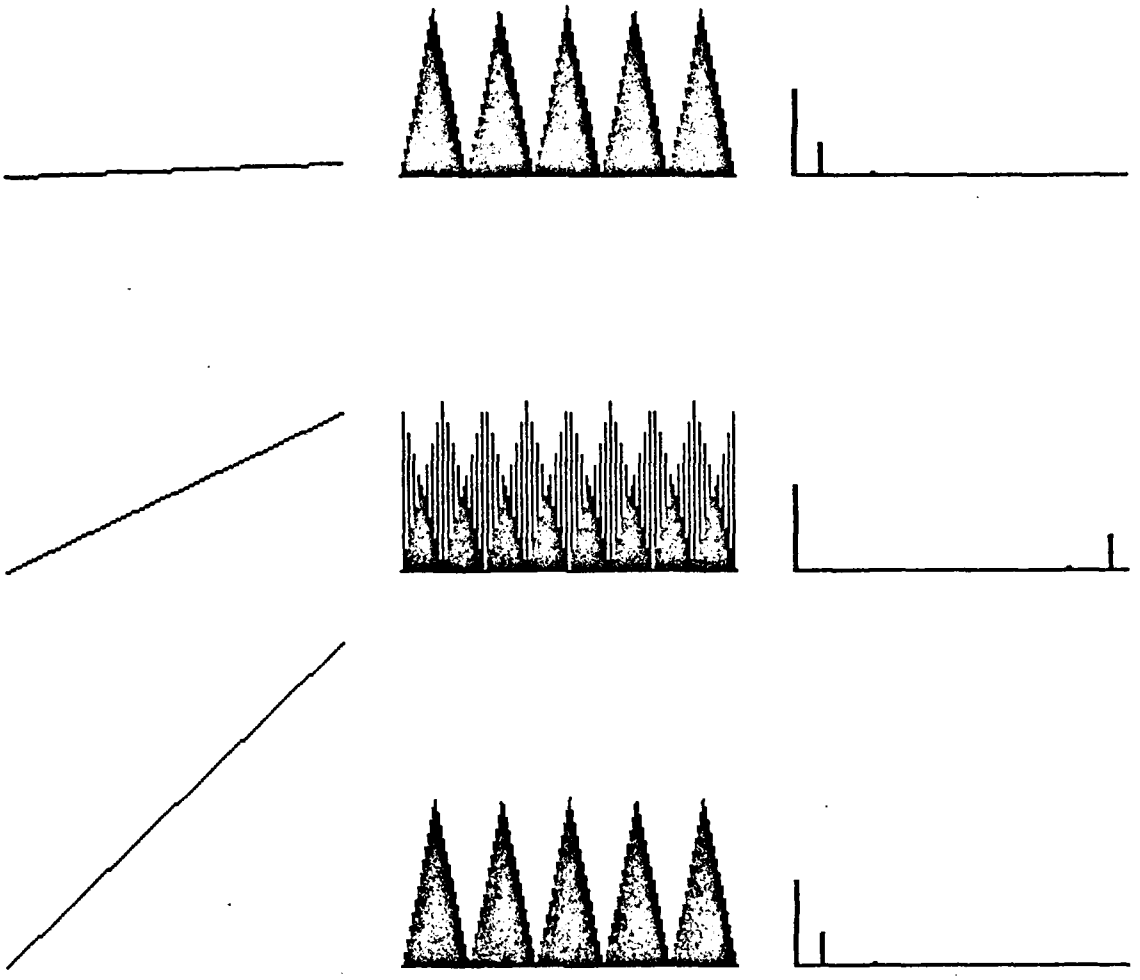


Figure 5-2: Three different lines, their error distributions, and the Fourier Transforms of the error distributions.

5.3. Bit-map lines using precomputed strokes

One obvious way to speed up the line drawing process is to compose longer lines using shorter precomputed segments. The line drawing algorithms now have to determine which of the set of segments to use and where to place them. This technique is particularly useful for displays where lines have to be approximated using "characters" [Jordan 74], or in display architectures which allow the updating of several pixels at a time, the subject of this thesis.

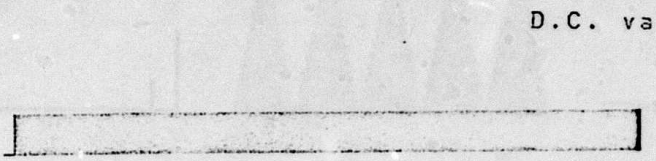


Figure 5-3: Sum of Fourier transforms for all lines with $dy = 128$; $0 \leq dx \leq dy$

Figure 5-4 shows an example line which has been drawn using two different strokes, each of which is eight pixels long. In general we can attempt to draw lines using N-pixel strokes. For shallow lines with slopes less than one, the N pixels of each stroke will lie in N different columns; conversely, for the steeper lines having slopes greater than one, the N pixels of each stroke will be in N different rows. In order to be able to update the display memory with the contents of the stroke, the memory has to have the ability to update either N rows or N columns at a time. This ability is provided by the NxN square memory organization.

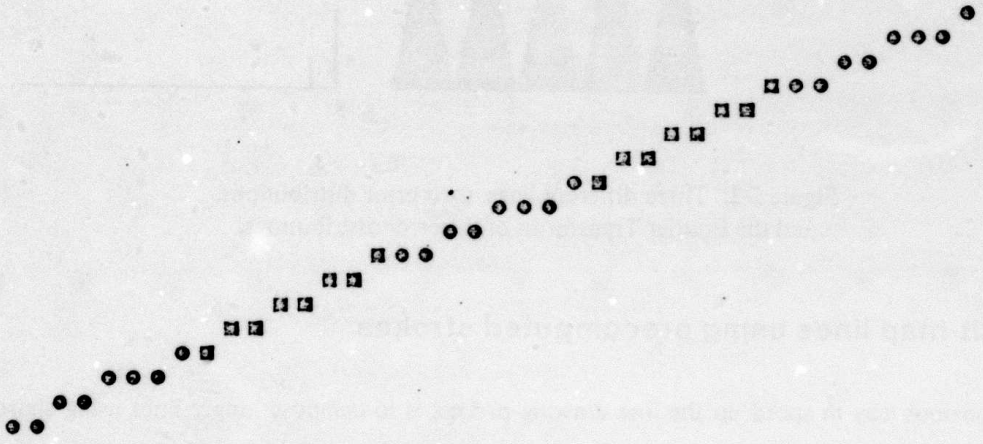


Figure 5-4: Line drawn using two different strokes.

We shall assume the ability to place the stroke at an arbitrary pixel boundary of the display. This implies that all strokes can defined such that the first pixel displayed is located at the origin (0,0) and the translation of every point of the stroke can be used to position the stroke at any pixel boundary.

We shall once again restrict the discussion to lines in the first octant. Lines in the other octants can be drawn either by using a combination of mirroring and transposing the strokes, or by using a larger set of strokes.

Strokes in the first octant have the same properties as the lines in the first octant. Each stroke extends over N columns, with each of the columns containing exactly one pixel. Each stroke can hence be defined as an array of integers *stroke*[0..N-1], such that the ith element of array defines the y position of the pixel in the ith column of the stroke. This stroke can then be positioned at any origin (ox,oy) using the following procedure.

```
for x := 0 to (N-1) do begin
   display(ox+x,oy+stroke[x])
```

Lines of arbitrary length can be drawn by truncating the final stroke to the desired length.

A line drawing algorithm which draws lines using strokes does so by repeatedly choosing one from a set of strokes and placing it at some position along the line. The algorithm also determines the total number of strokes that have to be precomputed and stored. This number can be affected by the primitives provided by the display; we have seen the effect of the presence or absence of mirroring and transposition. The algorithm also determines the maximum number of strokes used to draw any line.

Figure 5-5 shows two lines drawn using copies of one stroke only. The first 8 pixels of the line are used repetitively to generate the line. These lines show jumps and non-monotonicities. These occur because of the fact that the slope of the line might be either less than or greater than the slope of the stroke and repetitive use of this stroke will result in either undershooting or overshooting the line. This error will then have to be corrected by either jumping up or going back down again. The lines produced in this manner are hence unacceptable.

One alternative is to use two strokes, one of which is slightly shallower than the line desired and one which is slightly steeper. The two obvious ones are the ones which rise to the points just below and just above the y-intercept of the line in the Nth column (see Figure 5-6). Repeated use of these two strokes avoids the gap and monotonicity problems. Figure 5-7 shows a couple of lines drawn using two strokes for each line. The algorithm now has to determine which of the two strokes to place and where. There are two different ways in which this can be done. They differ in the amount of computation performed and also result in slightly different lines. The first method uses an algorithm that generates points along the line that are N pixels apart and then joins them using the precomputed strokes. Hence this method would generate x_0, x_N, x_{2N}, x_{3N}, - - -. The first line in

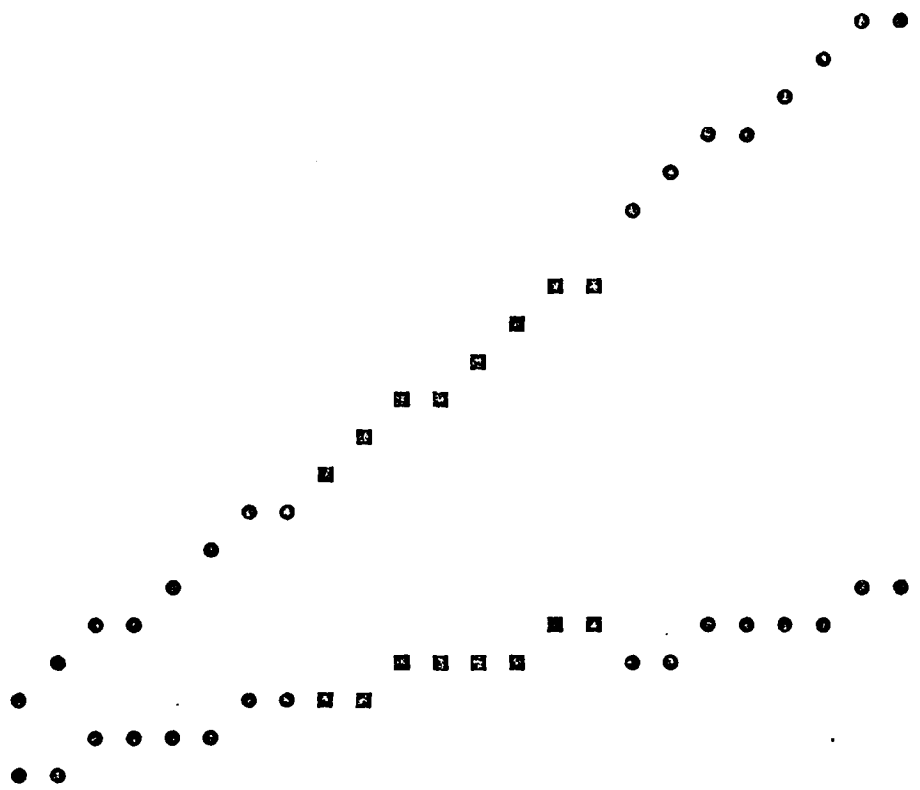


Figure 5-5: Line drawing using only one stroke.
The first line shows jumps, and the second shows non-monotonicities.

Figure 5-7 marks all such pixels. The second method also uses N-pixel strokes but generates both the endpoints of each stroke which are $(N-1)$ pixels apart. It will hence generate x_0, x_{N-1}, x_N, x_{2N-1}, x_{2N}, x_{3N-1} - - - (marked in the second line in Figure 5-7). The algorithm used in the second method is computationally more expensive because it has to generate more points, but it does result in slightly better lines.

5.3.1. The N-step algorithm

We can modify Bresenham's algorithm to generate every Nth point of a line. The line can then be drawn by joining these points using the precomputed strokes. Although there are $N+1$ points in the stroke that joins two points that are N apart, we can get by using an N-point stroke if we assume that the last point will be updated as the first point of the next stroke. Figure 5-8 shows the $N+1$ strokes that are needed to join such a tuple of points. The last point does not have to be stored as part of the stroke if we assume the discussed convention.

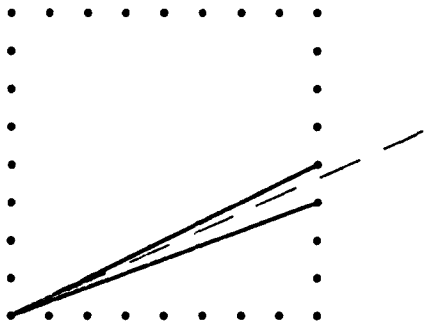


Figure 5-6: Two possible strokes for a given line.

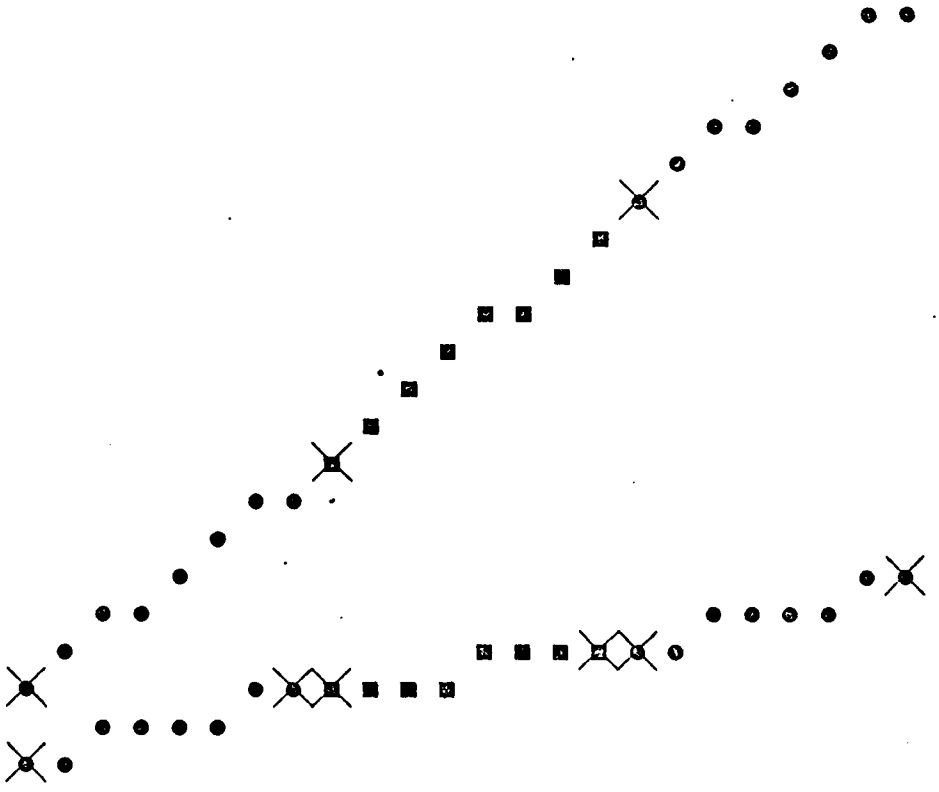


Figure 5-7: Lines drawing using two different strokes.

The following obvious algorithm generates every *N*th point of any given line.

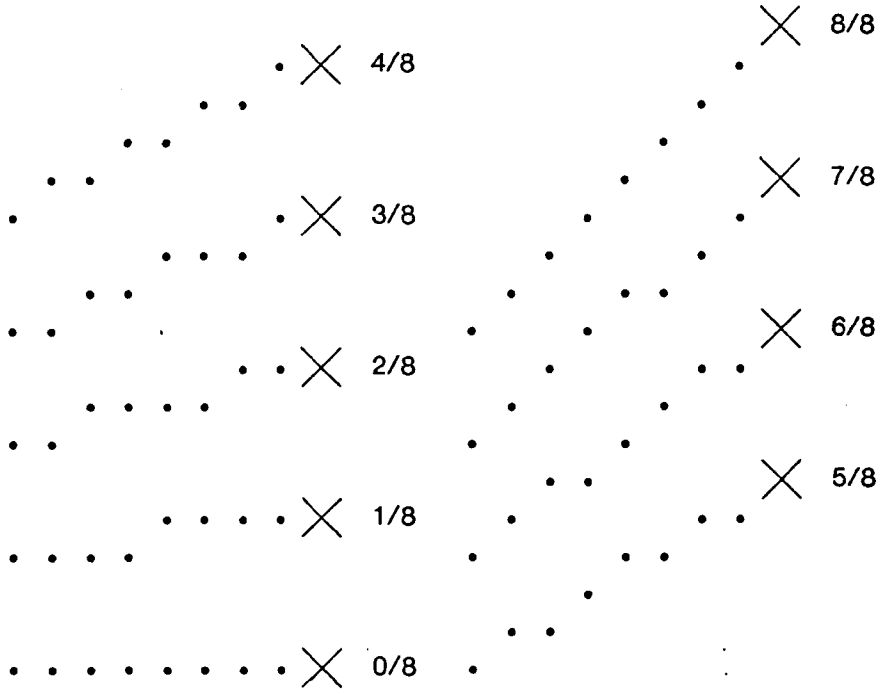


Figure 5-8: Strokes used by the N-step algorithm for $N=8$.

```
for x_i := 0 to dx by N do begin
    y_i := round( (dy/dx) * x_i)
    display(x_i,y_i)
end
```

Similar to the single step algorithm, this algorithm can be made to compute y_i incrementally by observing that the value of y_i will either be incremented by s ($= \lfloor N\,dy/dx \rfloor$) or by $s+1$ ($= \lfloor N\,dy/dx \rfloor + 1$) for each increment of x_i. This can easily be seen by examining all possibilities for the expression (round($(dy/dx)(x_i+N)$) $-$ round($(dy/dx)x_i$). There are four possibilities for the expression depending upon whether the subexpressions get rounded up or down. In all cases except one the value of the expression is s. In the case when the the first subexpression gets rounded up and the second subexpression gets rounded down the value of the expression is $s+1$. The incremental algorithm can be stated as the following.

```
y_i := 0;
e_i := N*(dy/dx);
for x_i := 0 to dx by N do begin
    display(x_i,y_i);
    if (e_i ≥ (s+1/2)) then begin
        y_i := y_i + s + 1;
        e_i := e_i + N*(dy/dx) - 1;
        end
    else begin
        y_i := y_i + s;
        e_i := e_i + N*(dy/dx);
        end
    end
```

Substituting $r = 2(e_i - s - 1/2)dx$ transforms this algorithm into the following.

```
y_i := 0;
r := 2*N*dy - 2*s*dx - dx;
for x_i := 0 to dx by N do begin
    display(x_i,y_i);
    if (r ≥ 0) then begin
        y_i := y_i + s + 1;
        r := r + (2*N*dy - 2*(s+1)*dx);
        end
    else begin
        y_i := y_i + s;
        r := r + (2*N*dy - 2*s*dx);
    end
end
```

This algorithm generates every Nth point of a given line. It can be used to draw the line if the appropriate stroke is displayed at every point generated. The stroke displayed depends upon the vertical displacement to the next point along the line, which we know to be either s or $s+1$. The line can now be drawn using the following algorithm. Also replaced is a precomputed value $a := 2N\,dy - 2s\,dx$.

```
y_i := 0;
r := a - dx;
for x_i := 0 to dx by N do begin
    if (r ≥ 0) then begin
        Display Stroke of height s+1 at (x_i,y_i);
        y_i := y_i + s + 1;
        r := r + a - 2*dx;
        end
    else begin
        Display Stroke of height s at (x_i,y_i);
        y_i := y_i + s;
        r := r + a;
    end
end
```

Although the values of s and a can be computed using multiplications and divisions, the following incremental algorithm computes their values in a manner similar to the Bresenham's algorithm. This can be hence used as a prologue to the line drawing algorithm above.

```
s := 0;
a := 0;
r := dy - dx;
for i := 0 to (N-1) do begin
    a := a + 2*dy;
    if (r ≥ 0) then begin
        s := s + 1;
        a := a - 2*dx;
        r := r - (dx - dy);
        end
    else r := r + dy;
end
```

Although this line drawing technique is faster than the single-bit algorithms, especially for architectures that allow parallel updates, it does not produce optimal lines. The N spaced points that it generates are optimal and hence the origin of each stroke can have a maximum error of only 1/2. However, there can be another error of 1/2 within a stroke which could be in the same direction as the stroke positioning error. However, the resulting total error at a point within a stroke can never add up to exactly 1 although it can be arbitrarily close to it. A proof to this effect is provided in Appendix A to this chapter. An example of an error of 0.9 for a line drawn using strokes is shown in Figure 5-9. It shows that for a line with $dx = 100$ and $dy = 85$, the value of $s = 6$, and the error can be observed at $x = 66$. Table 5-1 tabulates the maximum error for all possible lines with $dx \leq 128$ for different values of N.

Although this line drawing technique produces non-optimal lines, the end-point of the line is always exact. This is due to the fact that the maximum error at any point can never be greater than or equal to one. Hence any point that will lie exactly on the line (the end-point is one such point) will be computed correctly by this algorithm because an incorrect computation would imply an error greater than or equal to one. Exhaustive simulations of the algorithm also verify that the end-points are always correct and all lines also satisfy all other acceptability criteria.

Figure 5-10 shows two lines, the first of which is the optimal line drawn using the single-bit algorithm; the second is drawn using the N-step algorithm where $N=8$. The second line has a wavy behaviour because of the interaction between the slope of the line and the size of the stroke. This behaviour is the main reason for the unacceptability of this line drawing algorithm. The Fourier Transform measure for these lines shows undesirable peaks at the beat frequencies. This can be seen

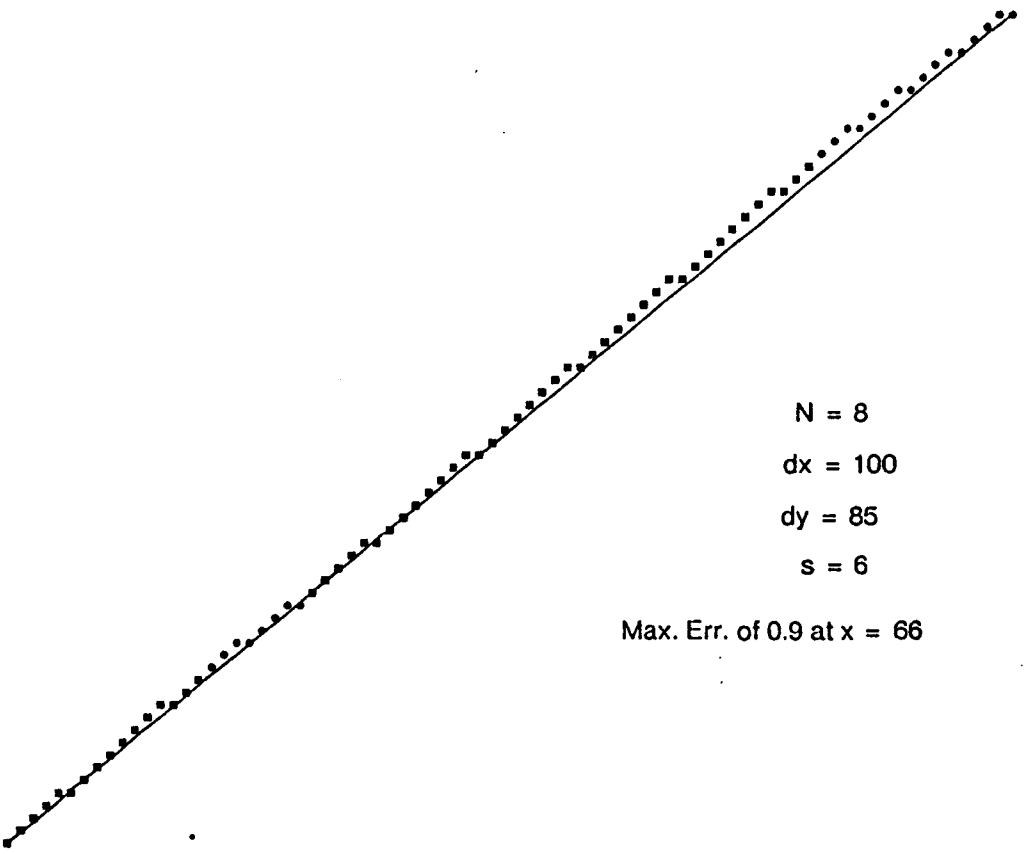


Figure 5-9: The N-step algorithm for $N=8$ results in a maximum error of 0.9 in this example.

N	Maximum error
1	0.5
2	0.992
3	0.828
4	0.992
5	0.894
6	0.992
7	0.923
8	0.992
9	0.938
10	0.992
11	0.950
12	0.992

Table 5-1: Maximum vertical error for different values of N in the N-step algorithm.

clearly when we sum the Fourier Transforms for lines with all frequencies. Figure 5-11 shows this

plot and we can see peaks at certain frequencies, which correspond to the undesirable ripples in the lines. Later in this chapter we shall discuss other stroke drawing algorithms which use more strokes and reduce the undesirable ripple effect.

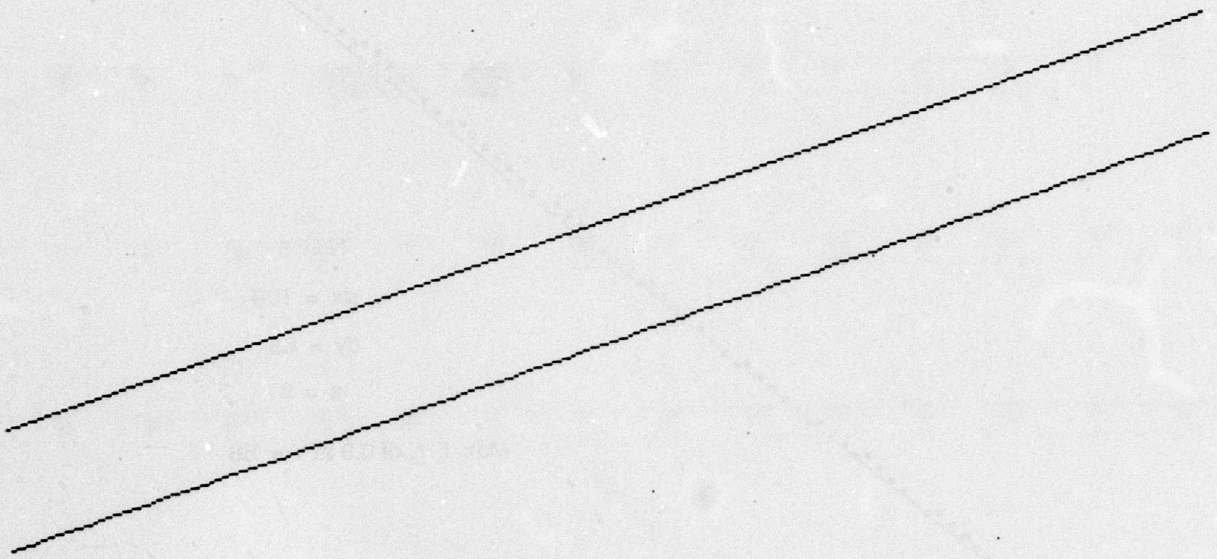
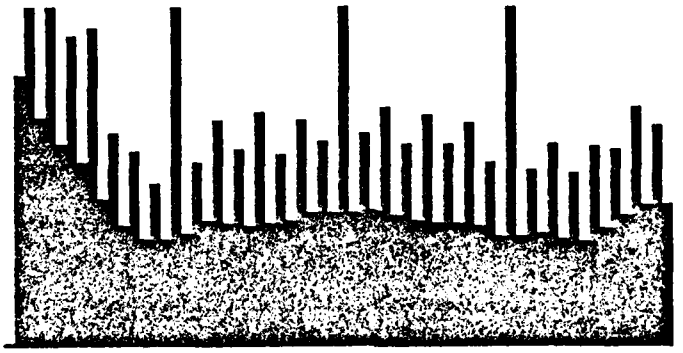


Figure 5-10: The first line uses a single-bit algorithm, the second line uses the N-step algorithm with $N=8$.

5.3.2. The (N – 1)-step algorithm

Since the end-points of an N point stroke are spaced $N-1$ points apart, we could compute points along the line which would correspond to the actual end-points of the stroke and join these points with the appropriate stroke. This would involve computing the locations of the x_0 and x_{N-1} as the end-points of the first stroke, the locations of x_N and x_{2N-1} for the location of the second stroke and so on. Notice that a line drawing algorithm of this variety would be computing twice as many points as the N-step algorithm. The $(N-1)$-step algorithm uses N strokes (Figure 5-12) as opposed to the $N+1$ used by the N-step algorithm.

The following obvious algorithm can be used.



```
D.C. value = 80.28
```

Figure 5-11: Sum of Fourier Transforms for the N-step algorithm for $N=8$.

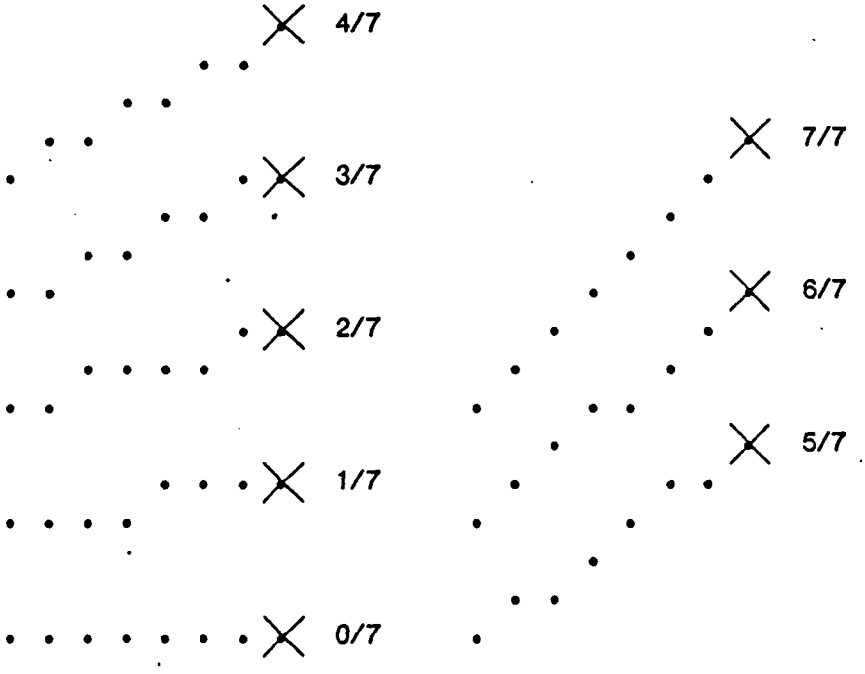


Figure 5-12: Strokes used by the $(N-1)$-step algorithm for $N=8$.

```
for x_i := 0 to dx by N do begin
    y_i := round( (dy/dx) * x_i);
    y_{i+N-1} := round( (dy/dx) * (x_{i+N-1}));
    Display stroke of height (y_{i+N-1} - y_i)
        at (x_i,y_i);
end
```

If s is defined to be $\lfloor (N-1)dy/dx \rfloor$, then by the same argument used for the N-step algorithm, the only two strokes used by any given line will have a height of either s or $s+1$. An equivalent incremental algorithm can then be stated as the following.

```
y_i := 0;
e_i := (N-1)*dy/dx;
for x_i := 0 to dx by N do begin
    if (e_i ≥ (s+1/2)) then begin
        Display stroke of height (s+1) at (x_i,y_i);
        y_i := y_i + s + 1;
        e_i := e_i + (dy/dx) - s - 1;
        end
    else begin
        Display stroke of height s at (x_i,y_i);
        y_i := y_i + s;
        e_i := e_i + (dy/dx) - s;
        end
    if (e_i ≥ 1/2) then begin
        y_i := y_i + 1;
        e_i := e_i + (N-1)*dy/dx - 1;
        end
    else
        e_i := e_i + (N-1)*dy/dx;
    end
```

This can be transformed into the following integer algorithm, where $a = (2N-1)dx - 2sdx$. The values of s and a can be computed using a prologue similar to the one in the N-step algorithm.

```
y_i := 0;
r := a - dx;
for x_i := 0 to dy by N do begin
    if (r ≥ 0) then begin
        Display stroke of height (s+1) at (x_i,y_i);
        y_i := y_i + s + 1;
        r := r + 2*dy - 2*dx;
        end
    else begin
        Display stroke of height s at (x_i,y_i);
        y_i := y_i + s;
        r := r + 2*dy;
        end
    if (r ≥ 0) then begin
        y_i := y_i + 1;
        r := r + a - 2*dx;
        end
    else
        r := r + a;
    end
```

The $(N-1)$-step algorithm can result in some point with an error of 1 in the case when N is odd. A proof for this is contained in Appendix B. In this case the algorithm will generates some lines with incorrect end-points which makes the algorithm unacceptable for odd values of N. Table 5-2 tabulates the maximum error for all possible lines with $dx \leq 128$ for different values of N.

N	Maximum error
1	0.5
2	0.5
3	1.0
4	0.831
5	1.0
6	0.898
7	1.0
8	0.927
9	1.0
10	0.942
11	1.0
12	0.953

Table 5-2: Maximum vertical error for different values of N.

For even values of N this algorithm produces slightly better lines than the N-step algorithm. This can be observed in the lines in Figure 5-13. The better appearance is verified by the sum of the Fourier transforms for all possible lines with $dx = 128$. Figure 5-14 plots the Fourier transform and a comparison with Figure 5-11 shows that this algorithm results in slightly lower extra ripple frequencies.

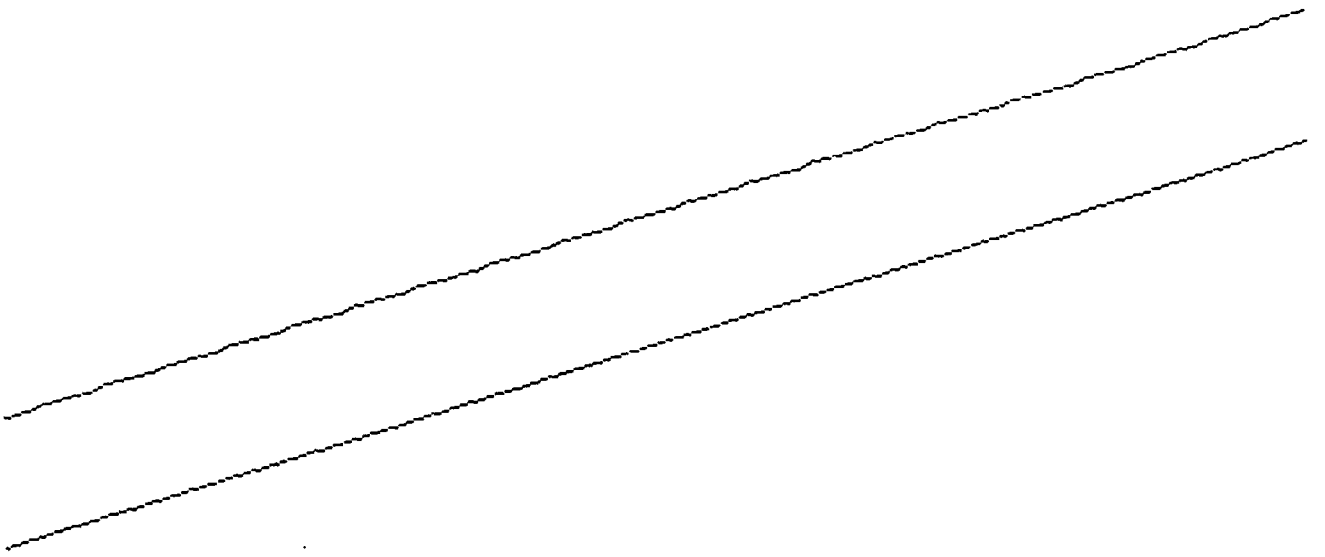


Figure 5-13: The first line uses the single-bit algorithm; the second uses the $(N-1)$-step algorithm for $N=8$.

```
D.C. value = 72.87
```

Figure 5-14: Sum of Fourier Transforms the $(N-1)$-step algorithm.

5.4. Better lines using a larger number of strokes

One way to explain the results of the two-stroke algorithms is to observe that only a single stroke is displayed for each rise in y, while the optimum line uses one of several strokes with the same rise. Figure 5-15 shows three of the eight different strokes possible for a stroke size of eight and rise of one for a horizontal jump of eight. The stroke used for the optimum line depends upon the slope of the line and the actual offset of the line at the origin of the stroke. For the three strokes shown in Figure 5-15, the first stroke would be optimal for a line with a slope of 1/8 and an offset of 0 at the beginning of a stroke. The second stroke would be optimal for a slope of 1/8 and an offset of 1/8, the third stroke for a slope of 1/10 and an offset of 0.

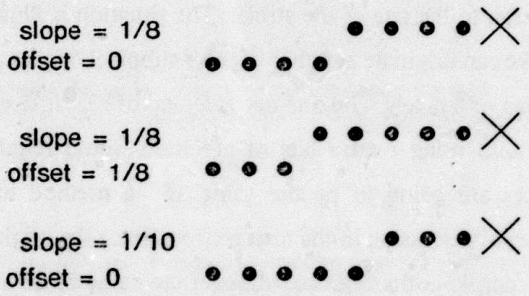


Figure 5-15: Three different strokes having the same rise.

This suggests a possible method for improving the lines produced by the stroke drawing algorithms. We could compute y to a higher precision at the N spaced points, and then use the sub-pixel offset and the higher-precision value of the slope available to choose from a larger set of strokes. As an example, one extra bit of precision would provide us with half-pixel y position for every x spaced N pixels apart. The N-step algorithm that would provide i extra bits of precision would be the following where $j = 2^i$, $s = \lfloor jN\,dy/dx \rfloor$, and $a = 2jN\,dx - 2js\,dx$. A prologue similar to the prologue in the N-step algorithm could be used to compute these values.

```
y_i := 0;
r := a - j*dx;
for x_i := 0 to dx by N do begin
    if (r ≥ 0) then begin
        Display Stroke of height s+1 at (x_i,y_i);
        y_i := y_i + s + 1;
        r := r + a - 2*j*dx;
        end
    else begin
        Display Stroke of height s at (x_i,y_i);
        y_i := y_i + s;
        r := r + a;
        end
    end
```

In this algorithm both y_i and s are represented with an integer part and an i bit fractional part. The fractional part of y_i is the value of the subpixel positioning at the origin of the stroke and the fractional part of s adds precision to the rise of the stroke. The situation is illustrated in Figure 5-16 for $N = 8$ and $i = 2$. The stroke can originate at either of the 4 subpixel origins and can have a rise of $0 \le s \le 32$ in the horizontal span of 8 pixels. The line needs a total of 132 different strokes. In general an N-step line drawing algorithm using i extra bits of precision would require a total $(N2^i+1)2^i$ strokes. Some of these strokes are going to be the same and a method to reduce the storage requirements will be the subject of discussion in the next section. Since the origin of the stroke will be the rounded value of y_i (for example in the example above if the value of the lower of bits of y_i is 2 then the first pixel of the stroke will be located at $y_i/4+1$), we can modify the algorithm to add a constant to the initial value of y_i and then merely use the integer part of y_i as the origin of the stroke. The modified algorithm is the following.

```
y_i := j/2;
r := a - j*dx;
for xi := 0 to dx by N do begin
    if (r ≥ 0) then begin
        Display Stroke of height s+1, offset y_i%j
                at (x_i,y_i/j);
        y_i := y_i + s + 1;
        r := r + a - 2*j*dx;
        end
    else begin
        Display Stroke of height s, offset y_i%j
                at (x_i,y_i/j);
        y_i := y_i + s;
        r := r + a;
        end
    end
```

Figure 5-17 shows the strokes used when $N = 8$ and $i = 1$. The use of extra precision in the

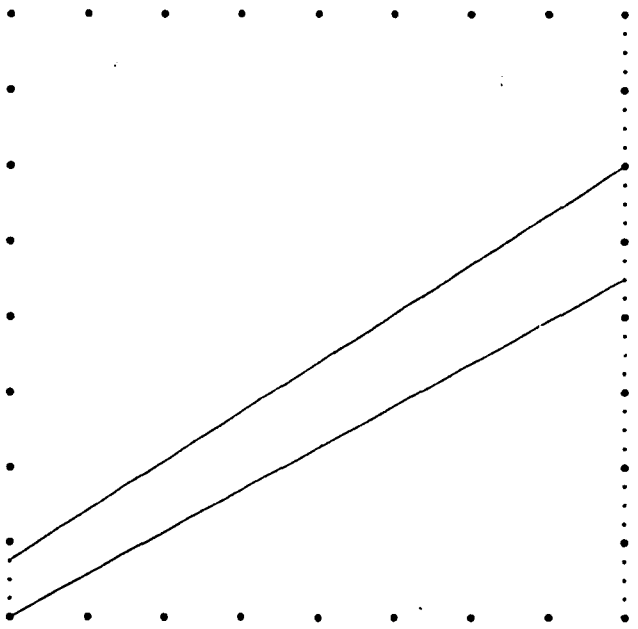


Figure 5-16: Sub-pixel positioning and extra precision rise.

positioning of the strokes helps decrease the maximum error in the lines produced. Since the individual strokes are placed to a precision of $1/j$ of a pixel, the stroke placement error can be a maximum of $1/(2j)$. Within the stroke each pixel can be also placed to a precision of $1/j$ and will then be rounded to the nearest pixel. The pixel placement produces a maximum error of $1/(2j)$ and the rounding produces an error of $1/2$. The total maximum error produced by using an algorithm with i extra bits of precision is hence $1/2 + 1/j$. This agrees with the simulations shown in Table 5-3 which tabulates the maximum error in the N-step algorithm against different values of i. Notice that increasing the precision would only make the lines asymptotically approach the optimum.

A better demonstration of an improvement in the appearance of the lines can be seen in Figure 5-18 which shows a line drawn using different amounts of precision. The sum of Fourier Transforms confirms the better appearance of the lines and is shown in Figure 5-19.

All the results of this section are also valid for the $(N-1)$-step algorithm. The $(N-1)$-step algorithm always produces slightly better lines than the N-step algorithm for a given amount of precision. But using higher precision is a better alternative than to use two steps in the inner loop as used by the $(N-1)$-step algorithm. My choice for the line drawing algorithm to be used would be the N-step algorithm with 2 extra bits of precision. Hence for $N = 8$, we would require a total of 132 strokes. Appendix C contains this complete algorithm including the stroke precomputing.

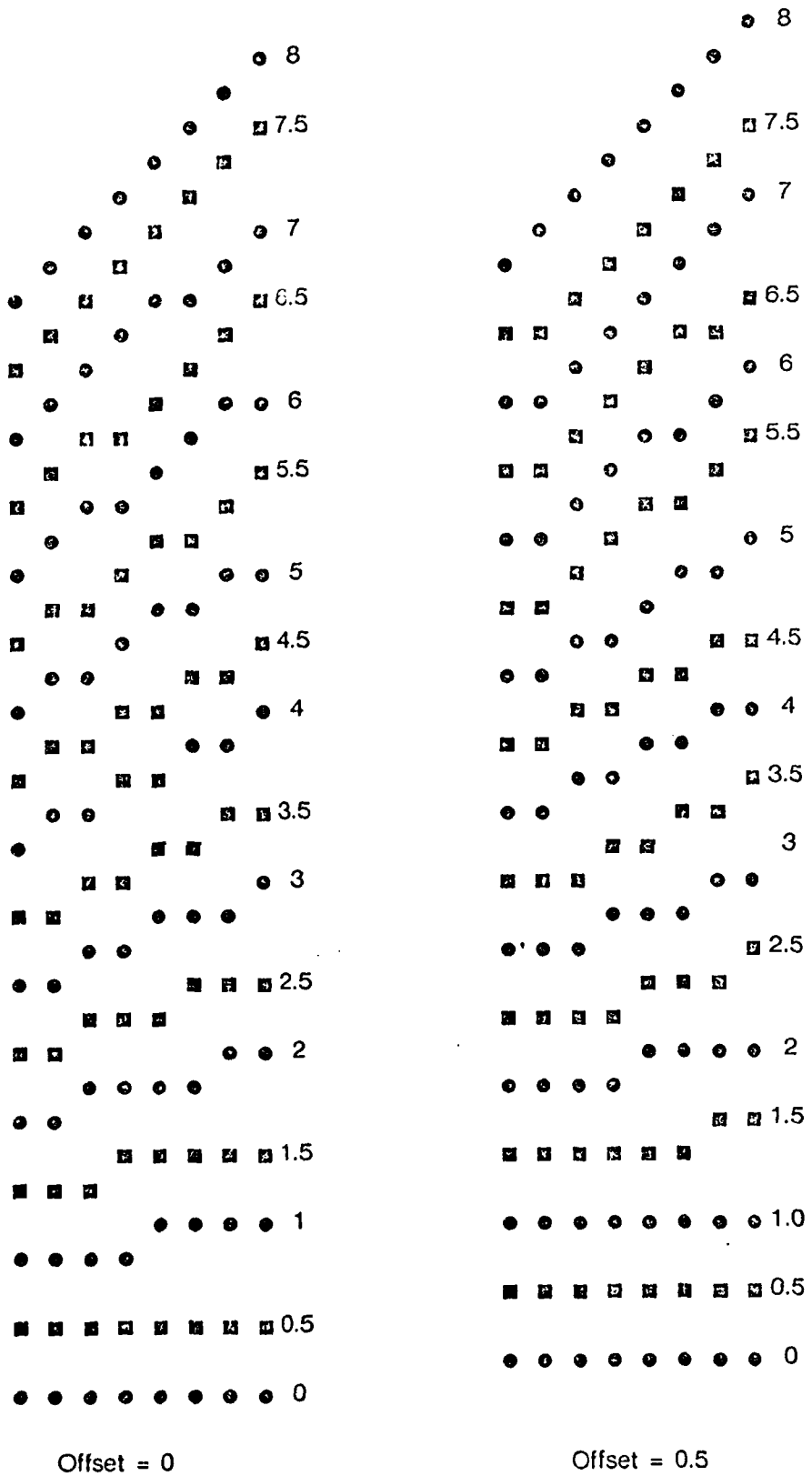


Figure 5-17: Strokes used in the N-step algorithm for $N=8$, $i=1$.

i	Maximum error
0	0.992
1	0.992
2	0.742
3	0.617
4	0.531
5	0.516

Table 5-3: Maximum vertical error for different values of i.

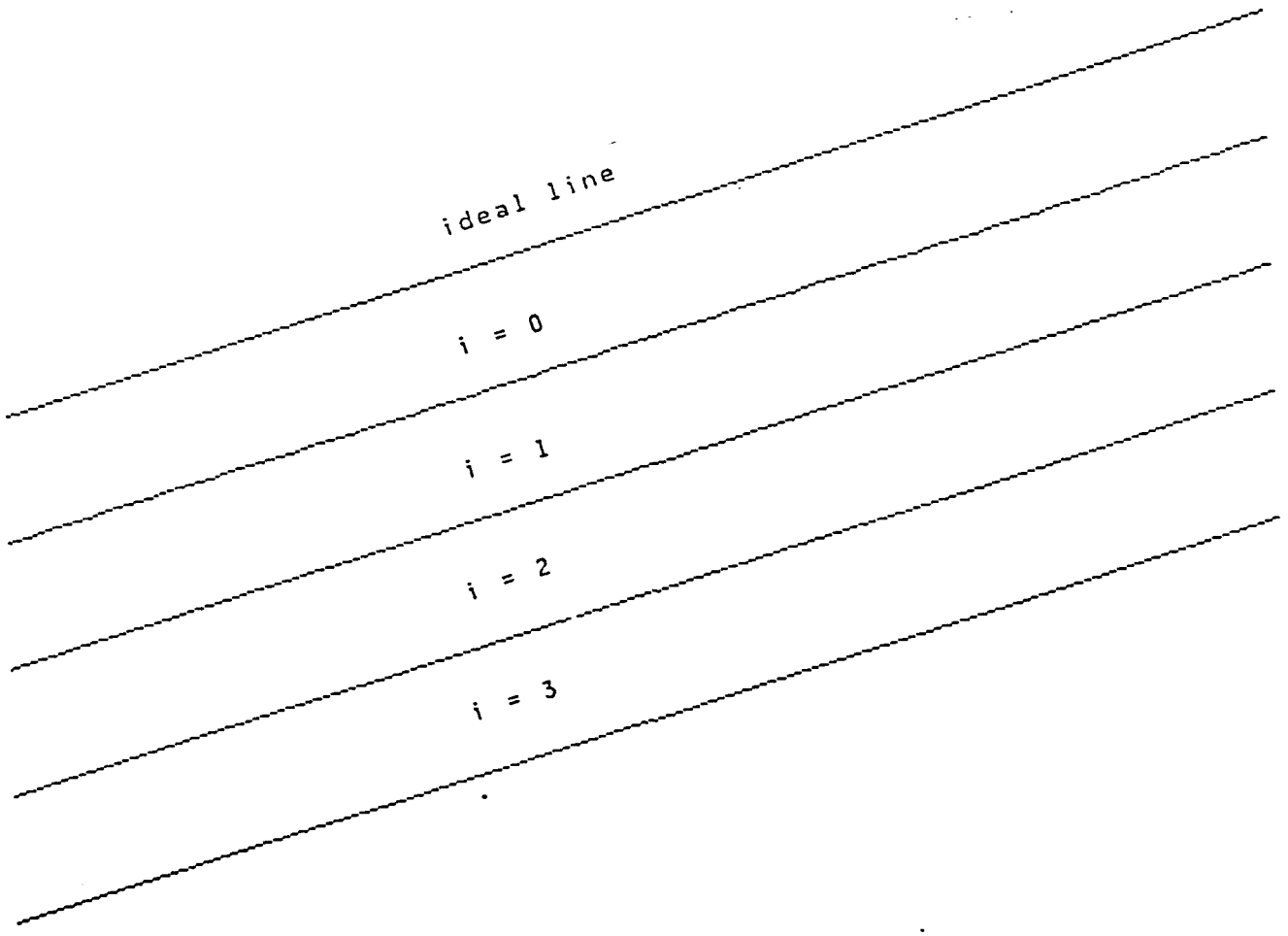


Figure 5-18: Line drawn with different amounts of extra precision.

5.4.1. Optimal lines using strokes

None of the stroke drawing algorithms we have discussed so far produce the optimal lines. The technique discussed in the previous section only approaches the optimum asymptotically but cannot achieve the optimum irrespective of the amount of precision used. This should not concern us to a

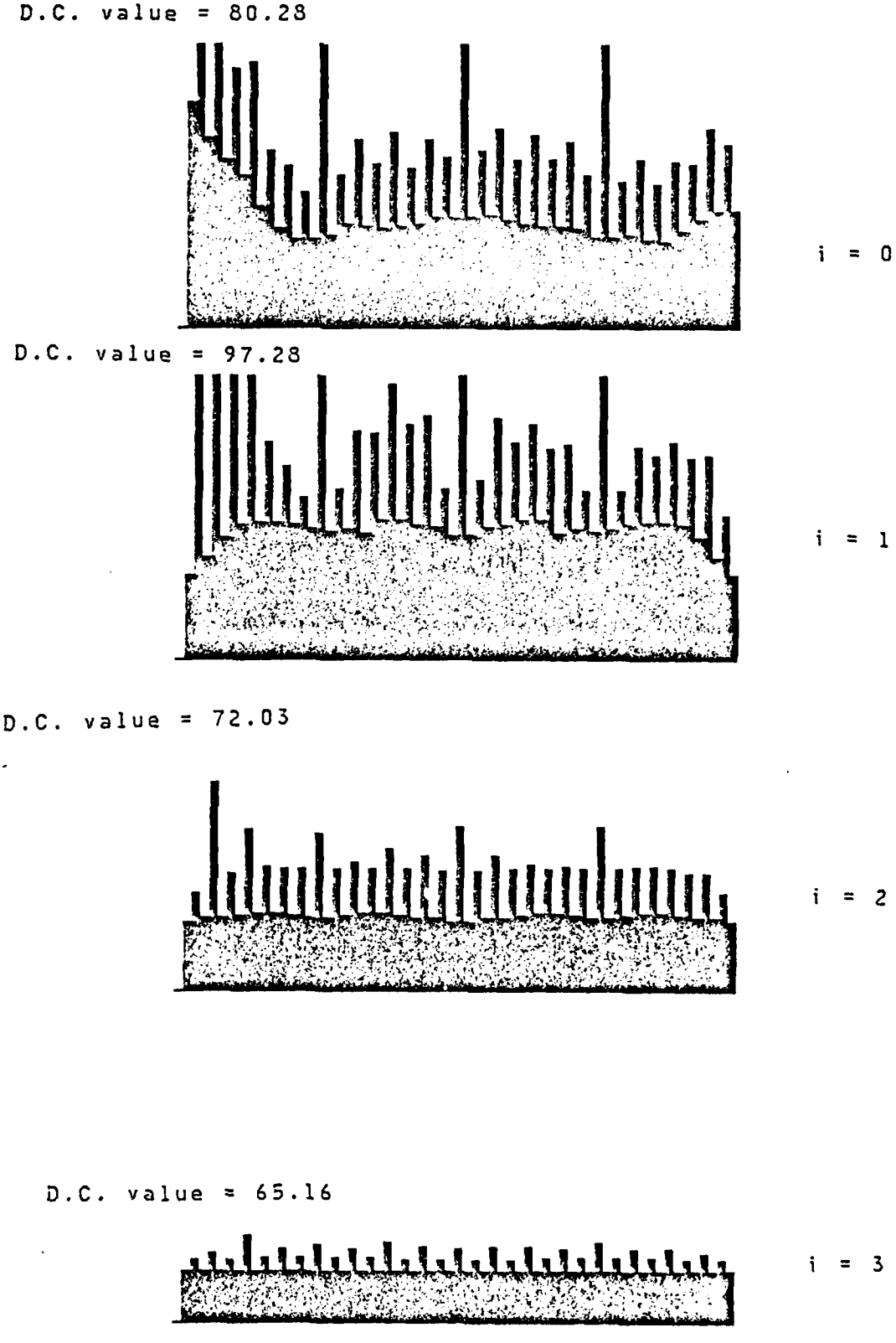


Figure 5-19: Sum of Fourier Transforms when different amount of extra precision is used.

large extent since one of the most popular line drawing algorithms (the DDA) also does not produce the optimum lines (Appendix D), but the lines produced are extremely good. This section is hence

only a theoretical study of how one can produce the optimum lines. In practice this technique will probably never be used.

The first question to ask is that if two strokes per line do not produce the optimum line, then what is the maximum number of strokes that a line might have to use in order to make it optimum. In order to determine this let us consider a line in the first octant with a rise of dy in dx. This line will illuminate a certain set of pixels in the first N columns. Let this set define stroke number 0 ($Stroke[0][0..N-1]$). Now imagine moving the line up slowly, or in other words imagine the line $y = (dy/dx)x + c$ with the value of c being slowly increased from 0 to 1. $Stroke[0]$ is the set of pixels illuminated when $c = 0$. A parallel set of pixels will be illuminated for $c = 1$, with the difference being that each point in this set will be one greater than in $Stroke[0]$. Hence for the increase of the value of c from 0 to 1, the N pixels in $Stroke[0]$ have all jumped up by one. This set of pixels will actually jump up one at a time as c is increased gradually from 0 to 1. This technique gives us N strokes and these can be used to generate the optimum line because these strokes represent all possibilities when a line of this slope originates at a non-integer pixel boundary. These strokes can be stored together with the values of c that each stroke was created with. The optimum line can now be drawn by computing every Nth pixel along the line together with the offset from the pixel below, lookup the closest value of the offset in the array of strokes and then use that stroke at that location. The r values of the Bresenham's algorithm provide a measure of the offset and can be used instead of the fractional values of c. This line drawing algorithm follows.

```
{ Generating the line using 8 strokes }

    { This computes s=floor(16*dy/dx) and a=16*dy-2*s*dx }
    s := 0; a := 0; r := dy-dx;
    for i := 1 to 8 do begin
        if (r ≥ 0) begin
            a := a+2*dy-2*dx;
            s := s+1;
            r := r+dy-dx;
            end
        else begin
            a := a+2*dy;
            r := r+dy;
            end
        end;

    { This computes the base stroke, which is the first eight
        pixels of the line and the error values associated with
        each of the pixels }
    err[1] := 2*dy-dx; line[0][0] := 0;
    for i := 1 to 7 do begin
        if (err[i] ≥ 0) begin
            line[0][i] := line[0][i-1]+1;
            err[i+1] := err[i]+2*dy-2*dx;
            end
        else begin
            line[0][i] := line[0][i-1];
            err[i+1] := err[i]+2*dy;
            end
        end

    { This computes the other eight lines and the displacement
        from the base line associated with each of them. }
    disp[0] := a-2*dx; disp[9] := 2*dxmax;
    for i := 1 to 8 do begin
        rmax := -2*dxmax;
        for j := 0 to 7 do begin
            line[i][j] := line[i-1][j];
            if (err[j+1] > rmax) begin
                rmax := err[j+1]; jmax := j;
                end
            end
        line[i][jmax] := line[i][jmax]+1;
        disp[i] := 2*dy-err[jmax+1]+a-2*dx;
        err[jmax+1] := -2*dxmax;
        end

   { This is the main loop which uses the 9 strokes to draw
      the line. }
    r := a-2*dx; y := 0;
    for i := 0 to dx/8 do begin
        j := 0;
        while (disp[j] ≤ r) j := j+1;
        jout := j-1;
```

```
            for j := 0 to 7 do begin
                array[i*8+j] := line[jout][j]+y;
            if (r ≥ 0) begin
                y := y+s+1;
                r := r+a-2*dx;
                end
            else begin
                y := y+s;
                r := r+a;
                end
            end
    end
```

The inner loop of this algorithm searches for the stroke to use and then uses that stroke. This algorithm is obviously inefficient for all lines with lengths less than N^2, because the N strokes that have to be computed contain N^2 pixels. Even for extremely long lines the inner loop would have to done extremely efficiently to make this algorithm worthwhile. Conventional processor implementations would require a longer time to search for the stroke than to write it into memory, especially if the memory allowed the updating of such a stroke in parallel.

5.5. Total number of strokes

We have studied several algorithms that draw lines using N-pixel strokes. These algorithms use different numbers of strokes to draw a line and hence require a different set of strokes to choose from. But for a given stroke size there is a maximum number of strokes that ever get used and all possible lines can be drawn using these strokes. Although at first sight it may seem that for N-pixel strokes with a rise of s there should be $N!/(s!\,N!)$ different possible strokes, this may not be the case because the jumps have to be spaced out evenly in order for the stroke to be part of a line. Hence, for example, a stroke that jumps four pixels in the first four pixels of an eight pixel stroke and then stays horizontal would never be used. Table 5-4 contains all the possible strokes for $N = 8$. Table 5-5 tabulates the total number of strokes required for different stroke sizes.

None of the algorithms we have discussed can choose the optimum stroke from this set. Some of the algorithms choose from a smaller set and some choose from a larger set. In either case the set used is a subset of the total number of strokes possible. In the case when the algorithm chooses from a larger set of strokes, the set will contain non-unique strokes. A simple way of avoiding the redundancy is to provide a table-lookup with pointers to one of the total set of strokes.

00000000	00000001	00000011
00000111	00001111	00011111
00111111	01111111	00111112
00111122	00111222	01111222
01112222	01111112	01111122
00001112	00011112	00011122
00011222	00112223	00112233
01112233	01122233	01112223
01122333	00011223	00111223
01122334	01122344	01123344
01223344	00112234	00112334
00122344	00122334	01123345
01123445	01223445	01223455
01223345	01233455	00122345
00123345	00123445	00123455
01123455	01223456	01233456
01234456	01234556	01234566
00123456	01123456	01234567

Number of strokes = 54

Table 5-4: All possible strokes for $N = 8$.

N	Number of strokes
2	2
3	3
4	8
5	14
6	24
7	36
8	54
9	76
10	104
11	136
12	178

Table 5-5: Number of all possible strokes for different values of N.

Appendix A

This appendix proves that the lines produced by the N-step algorithm always have a vertical error of less than 1, and hence the algorithm always generates lines with matching end-points. The proof is adapted from a similar proof by Mike Spreitzer of CalTech. This proof shall also assume that the line is drawn from (0,0) to (dx,dy) and that $dx \geq dy \geq 0$. We shall use z as the measure of vertical error for the line; z_i is the signed distance from the pixel at $x = i$ to the line. This distance can be computed as $y_i - i\,dy/dx$.

The N-step algorithm computes every Nth point of the lines and guarantees that $|z_{Ni}| \leq 1/2$. These points are joined with one of two strokes that represent a line of slope s/N, where s is the vertical distance from the origin of the stroke to the origin of the next stroke ($= y_{Ni+N} - y_{Ni}$). If the stroke is generated using the Bresenham algorithm aiming at a slope of s/N, then we have $|z_{Ni+j} - (j/N)(z_{Ni+N} - z_{Ni})| \leq 1/2$, for $0 \leq j \leq N$. We consider two cases: for when the expresson is positive and negative.

1. When the expression is positive we have $z_{Ni+j} \leq (j/N)(z_{Ni+N} - z_{Ni}) + 1/2$. From the triangle inequality, we know that $|z_{Ni+N} - z_{Ni}| \leq 1/2$. However, the equality case nevers occurs because if it did, then the slope of the line would an integer multiple of $1/N$ and the z_{Ni} would be zero for all i. So the triangle inequality can be restated as $|z_{Ni+N} - z_{Ni}| < 1/2$. For $1 \leq j \leq N/2$, this yields $z_{Ni+j} < 1$.

2. The negative case, by similar arguments yields that for $1 \leq j \leq N/2$, $z_{Ni+j} > -1$.

Both these arguments together result in $|z_{Ni+j}| < 1$ for $1 \leq j \leq N/2$.

Approaching from the other side (i.e. $x = Ni - j$), we can get that $|z_{Ni-j}| < 1$ for $1 \leq j \leq N/2$. Combining the two results we have $|z_{Ni+j}| < 1$ for $1 \leq j \leq N$ when N is even.

This proves the fact that the maximum vertical error at any point can never equal or exceed 1. This also proves the fact that the end-point shall be computed correctly because if the end-point was computed incorrectly then the error at the end-point would be equal to or greater than 1.

Appendix B

In the $N-1$ step algorithm the condition for the vertical error at any point along the line gets restated as $|z_{Ni+j} - j/(N-1)(z_{Ni+N-1} - z_{Ni})| \leq 1/2$. When N is odd, it is the step of combining $|z_{Ni+j}| < 1$ for $1 \leq j \leq N-1/2$, and $|z_{Ni-j}| < 1$ for $1 \leq j \leq N-1/2$, that does guarantee $|z_{Ni+j}| < 1$ for

$j = (N+1)/2$. So if $N = 7$, then the $N-1$ step algorithm does not guarantee that the error will be less than 1 at the mid-point of the stroke, that is when $j = 4$.

Figure 5-20 shows an example for $N = 3$ for a line with $dx = 10$ and $dy = 5$.

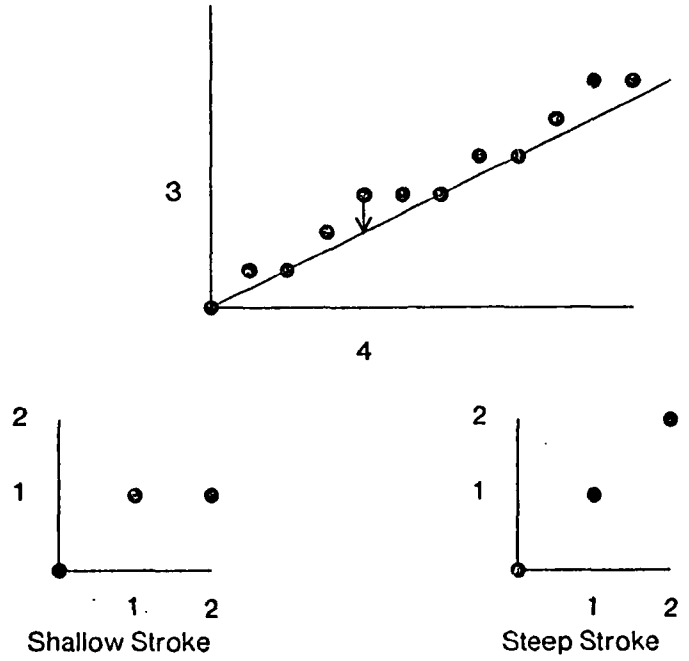


Figure 5-20: Error of 1 made by the $(N-1)$-step algorithm in a line with $dx = 10$, $dy = 5$, and $N = 3$.

Appendix C

This is the recommended line drawing algorithm for 8-pixel strokes. The first algorithm presented precomputes the necessary strokes. The algorithm presented uses multiplications, which does not matter at precomputing time, although the obvious incremental algorithm can be used.

```
{ Precomputing strokes }
for height := 0 to 32 do
    for offset := 0 to 3 do
        for x := 0 to 7 do
            Stroke (height,offset) := (offset + height*x/8)4;
```

The following algorithm uses the precomputed strokes to draw the line.

```
{ Line drawing algorithm }

{ Prologue }
s := 0; a := 0;
r := 4*dy - dx;
for i := 0 to 7 do begin
    a := a + 8*dy;
    if (r ≥ 0) then begin
        s := s + 1;
        a := a - 8*dx;
        r := r + 4*dy - dx;
        end
    else r := r + 4*dy;
end;

{ Line drawing }
y_i := 2;
r := a - 4*dx;
for x_i := 0 to dx by 8 do begin
    if (r ≥ 0) then begin
        Display Stroke (s+1,y_i%4) at (x_i,y_i/4);
        y_i := y_i + s + 1;
        r := r + a - 8*dx;
        end
    else begin
        Display Stroke (s,y_i%4) at (x_i,y_i/4);
        y_i := y_i + s;
        r := r + a;
        end;
end;
```

Appendix D

For lines drawn from (0,0) to (*dx*,*dy*) such that $0 \leq dy \leq dx$, the Digital Differential Analyzer (DDA) algorithm draws the line in the following manner.

```
y_i := 0; r := (dy/dx)/2;
for x_i := 0 to dx do begin
    display(x_i,y_i);
    if (r ≥ 1/2) then begin
        y_i := y_i + 1;
        r := r + dy/dx - 1;
        end
    else begin
        r := r + dy/dx;
        end;
    end;
```

Implementations of the DDA represent *dy*/*dx* as an integer with a fixed amount of precision. Because this rational number may be a repeated fraction, this representation can never represent some fractions without incorporating an error. The error can cause the diagonal or horizontal move

decision to take the wrong direction and result in an incorrect line. An example is when $dx = 12$, and $dy = 2$. The fraction 2/12 cannot be represented as an integer (because it is a repeated fraction of the form 0.0010101010101010....). Depending upon the amount of precision used the fraction will be either rounded up or down. In the case when it is rounded down, the value of y_i at $x_i = 9$ will be less than the actual 1.5, and the DDA will hence illuminate the pixel (9,1) which is incorrect. Rounding up can similarly cause a higher pixel to be illuminated. No amount of precision will generate correct lines, although the maximum error will indeed decrease with increasing amounts of precision.

Chapter 6
Filtering

Bit-map images show annoying visual effects in the form of "jaggies" or "staircases" when a shaded region jumps between rows or columns of pixels. On a display device that can present gray-scale images, this effect can be avoided by smoothing the jump using gray intensity values between white and black.

The jagged effect is due to the sampling of a sharp edge over a fixed grid. The original edge contains frequencies higher than those that can be faithfully reproduced by the samples (Nyquist's theorem). This results in higher frequencies aliasing as lower ones; the error is hence known as *aliasing*. Figure 6-1 shows a high and low frequency signal both of which result in the same samples. Low pass filtering of the signal before sampling can remove the aliasing problem; this preprocessing is hence called *anti-aliasing*. Figure 6-2 illustrates this situation in one dimension. The first part shows the sampling of the unfiltered signal and the second part shows the sampling of the filtered signal. As illustrated by the figure, the filtered samples require the ability to display gray intensity values. For the proper appearance of the resulting image the filter has to be properly chosen, else the images produced tend to show other annoying aliasing artifacts. In general, the filter should be a parameter of the anti-aliasing computations and could be varied until an acceptable image results.

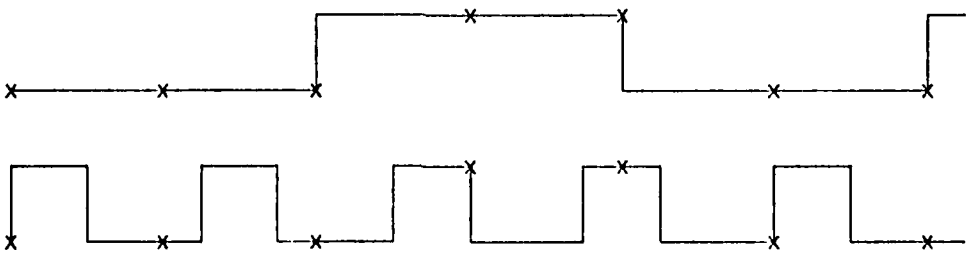


Figure 6-1: The aliasing problem.

The filtering of the image with a low-pass filter is equivalent to an averaging process: the intensity

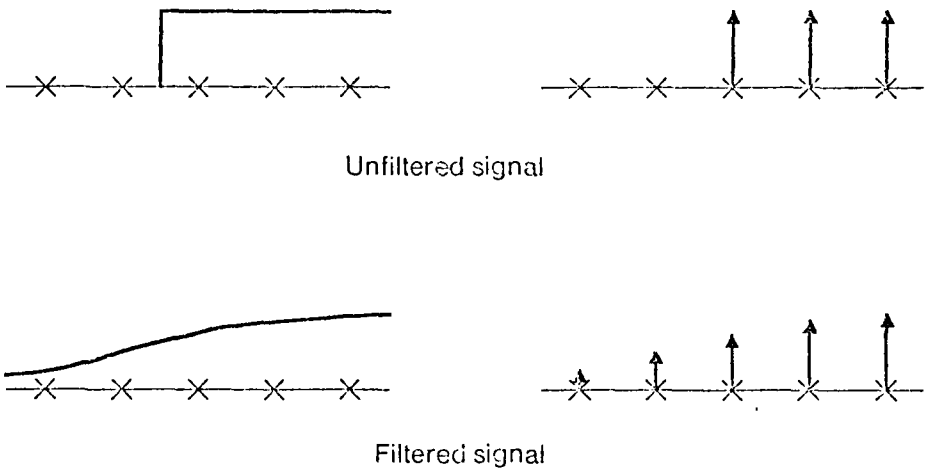


Figure 6-2: Sampling an unfiltered and a filtered signal.

of a sample is determined by the image brightness in the near vicinity of the pixel. The averaging function is the space-domain equivalent of the low-pass frequency domain filter and is known as the *filter function*. The filter function is also related to the spatial frequency distribution of the light emitted by a pixel on the display. Figure 6-3 shows a few filter functions together with their frequency spectrums. The first filter in the figure is the *impulse* function which corresponds to the images produced by the bit-map algorithms that exhibit the aliasing problems. The *sinc* filter has the ideal low-pass frequency response and results in samples that truly model the low-frequency characteristics of the sampled signal. The *triangular* filter, which is an approximation of the sinc filter is attractive for its mathematical tractability and is probably the most commonly used function. The *rectangular* filter is probably the easiest filter to implement because the averaging process involves only additions. The *Gaussian* filter is the closest approximation of the spatial distribution of the light emitted by a pixel. Apart from the shape the other important property of the filter function is its extent. All filters with a finite extent pass some of the frequencies beyond the Nyquist frequency. In practice both the shape and extent are varied until an aesthetically satisfactory image results.

In order to sample two-dimensional images the filter functions have to be extended to two dimensions. There are two different ways in which this can be done. In the first method the one-dimensional filter is rotated circularly to produce the two-dimensional filter. This results in a circularly symmetric filter which has some computational advantages. The other method extends the one-dimensional filter into two-dimensional by merely multiplying two one-dimensional filters. In practice either of the two techniques can be used, depending upon the convenience of the situation.

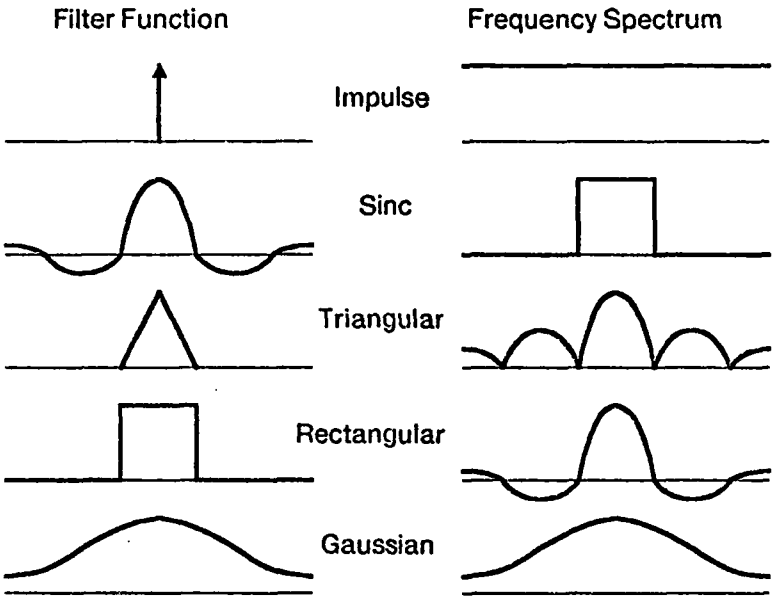


Figure 6-3: A few possible filter functions and their frequency response.

6.1. Computing filtered images

The most popular method for computing filtered images is to numerically integrate the image over the area of the filter to compute the intensity of each pixel in the image. The step size for the integration is determined by the accuracy desired in the intensity value. The flat square filter function which extends to the adjoining pixels (Figure 6-4) will be used to illustrate the problem. Other functions result in similar conclusions.

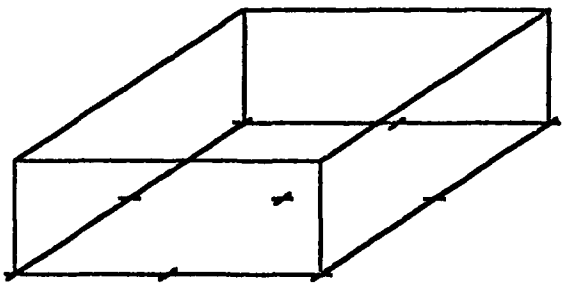


Figure 6-4: Flat square filter function.

If the pixel resulting from the averaging process is supposed to be accurate to 8 bits (i.e. 1 part in 256), then the image computing process should provide at least 256 equi-spaced samples (on a 16x16 grid) which can then be averaged to compute the intensity value. This implies that the image has to be computed to a higher resolution and then averaged to compute the filtered image. The resolution should be 8 times the actual resolution in order to compute an image with a resolution of 8 bits. This is the reason that an 8-bit gray-scale image is considered to be equivalent to one that has 8 times the resolution. Similarly an image with 4 bits of gray-scale would be equivalent to one with twice the resolution. Different filter functions result in approximately the same numbers for the image resolution desired. Experiments with the human visual perception confirm these results [Leler 80].

It is seldom necessary to compute a high resolution image because geometric information leads directly to the filtered intensity of the pixels. The following subsection illustrates this technique with an example.

6.1.1. Filtering straight edges

Throughout the rest of this chapter we shall assume the use of a conical filter function: the function has its maximum value at the center of a pixel and decreases linearly to zero at a distance r from the pixel center. The radius of the filter is r, measured using the convention that a unit distance is the distance between two adjacent pixel centers. The function is normalized so that the enclosed volume is 1. This function is the circular extension of the triangular filter. Figure 6-5 shows the filter function with radius 1. This filter is chosen both for its mathemetical properties and the fact that it is a close approximation to the ideal sinc function.

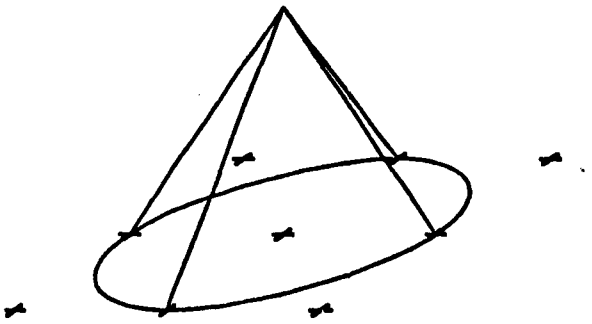


Figure 6-5: Conical filter function.

When a straight edge passes through a pixel, the pixel's intensity should be proportional to the volume of the cone intersected by the edge (see Figure 6-6). Because of the circular symmetry of the filter, only the perpendicular distance p from the pixel center to the edge is needed to determine the volume intersected. Thus we may write $I = F_e(p)$ where F_e is determined solely by the choice of the filter, provided it is circularly symmetric[4]. Note that $F_e(p) = 0$ for $p \leq -r$, and $F_e(p) = 1$ for $p \geq r$. For $-r \leq p \leq r$, $F_e(p)$ can be computed by numerically integrating the volume of the cone intersected by an edge at a distance p from the pixel center.

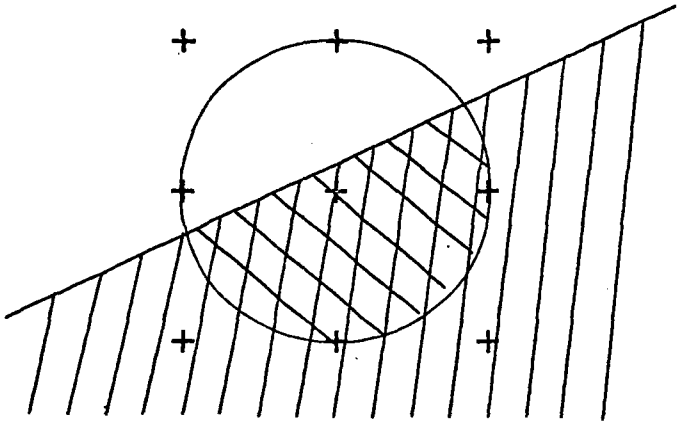


Figure 6-6: Edge intersecting a pixel.

Since I is represented as a g-bit integer which can take on only 2^g distinct values, the function F_e can be be converted into a simple linear table. If F_e is a linear function then this table would contain only 2^g entries; for a nonlinear function the number of entries is determined by the maximum value of the derivative of the function. Table 6-1 illustrates values of F_e for the conical filter with $r = 1$, if I has 4 bits of resolution. Because of the limited resolution in the range of the table, we can actually cut off the table at a domain slightly shorter than the theoretical maximum allowed by the function.

Table look-up makes the anti-aliasing computations more efficient. By performing the usual scan-conversion computations with slightly higher precision, we then use the sub-pixel position on an edge as an index into a table which will provide the shade value for a pixel. We have discussed one such table for single edges intersecting pixels. The next subsection shall discuss a universal table that can

[4]This discussion will assume that each object shall be intensified to the maximum intensity. Objects with lower intensities can be shown by merely scaling the intensities used in this discussion.

p	$F_e(p)$	p	$F_e(p)$
$\leq -12/16$	0	1/16	8
$-11/16$	1	2/16	9
$-10/16$	1	3/16	10
$-9/16$	1	4/16	11
$-8/16$	2	5/16	12
$-7/16$	2	6/16	12
$-6/16$	3	7/16	13
$-5/16$	3	8/16	13
$-4/16$	4	9/16	14
$-3/16$	5	10/16	14
$-2/16$	6	11/16	14
$-1/16$	7	$\geq 12/16$	15
0/16	8		

Table 6-1: Pixel intensities from the distance to edges.

be used for all polygonal geometries. It is benefical, however, to compile separate tables which can be used more efficiently for more frequently used geometries. One such example is line drawing. Intensities for pixels intersected by a line can be considered as the difference between the intensity contributions of the two edges of the line (Figure 6-7). Hence there are two parameters that are needed to determine the intensity: the thickness t of the line and the perpendicular distance p from the pixel center to the line center. Thus we may write $I = F_l(p,t)$, where F_l again is determined solely by the choice of the filter function. Table 6-2 illustrates values of F_l for $t = 1$ for the conical filter function of radius 1.

6.1.2. Universal Table for polygons

All polygons can be split up into trapezoids by drawing horizontal lines at each corner (Figure 4-5). It shall be assumed that polygons shall be formed by combining its subset of trapezoids. This section shall discuss the anti-aliasing tables needed to form trapezoids. To do so, we shall first examine the eleven possible ways in which a trapezoid can intersect a pixel.

1. The pixel is totally covered by the trapezoid (Figure 6-8). The intensity of such a pixel shall be the same as the shade of the polygon. Hence I = Max.

2. Only one side edge of the trapezoid intersects the pixel (Figure 6-9). Only the perpendicular distance from the center of the pixel to the edge is needed to determine the intensity of the pixel. Hence we can say that $I = F_e(p)$.

3. Only one horizontal edge of the trapezoid intersects the pixel (Figure 6-10). Once again only the perpendicular distance is needed, but in this case the perpendicular distance is the same as the vertical distance. If y is the position of the center of the pixel and $y1$ is the position of the edge then the perndicular distance is $y-y1$. Hence $I = F_e(y1-y)$.

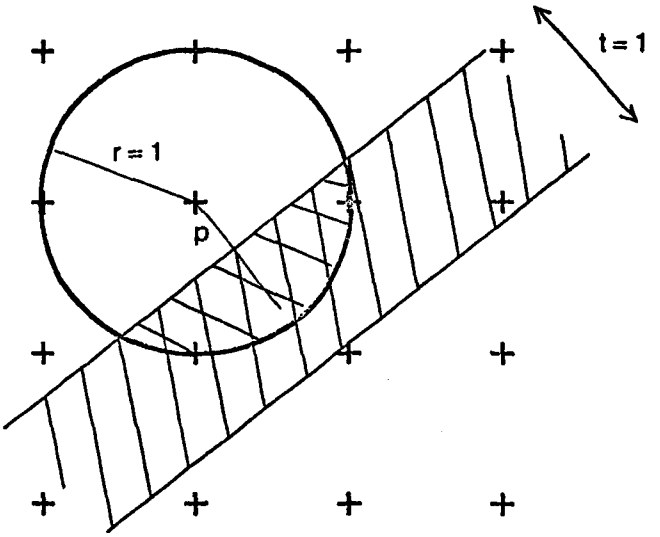


Figure 6-7: Line intersecting a pixel.

$\|p\|$	$F_f(p,1)$	$\|p\|$	$F_f(p,1)$
0/16	12	11/16	5
1/16	12	12/16	4
2/16	11	13/16	3
3/16	11	14/16	3
4/16	11	15/16	2
5/16	10	16/16	2
6/16	9	17/16	1
7/16	8	18/16	1
8/16	8	19/16	1
9/16	7	≥20/16	0
10/16	6		

Table 6-2: Pixel intensities from distance to lines.

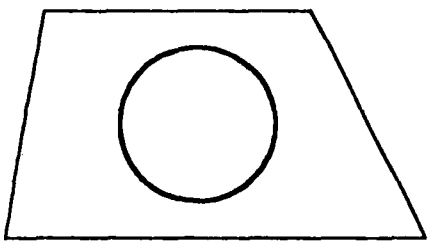


Figure 6-8:

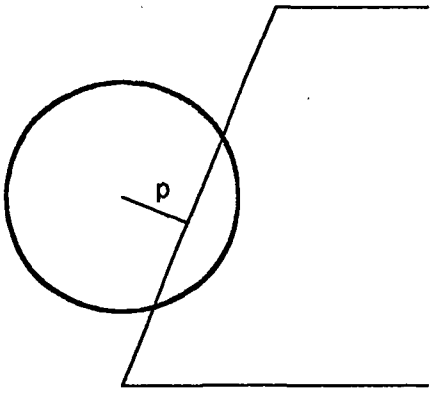


Figure 6-9:

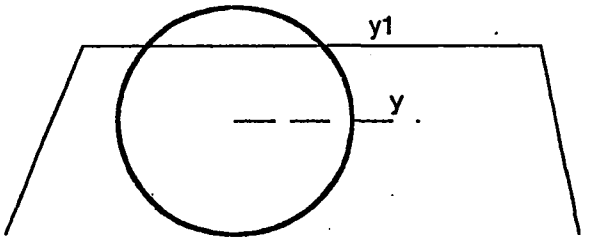


Figure 6-10:

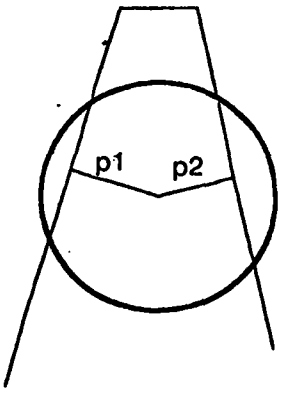


Figure 6-11:

4. Both the side edges of the trapezoid intersect the pixel (Figure 6-11). In this case the intensity of the pixel is the difference between the intensity conributions of the two edges. Hence $I = F_e(p1) - F_e(p2)$.

5. Both the horizontal edges of the trapezoid intersect the pixel (Figure 6-12).

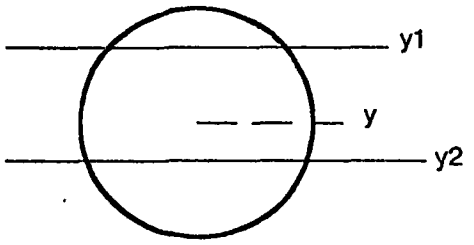


Figure 6-12:

As in the previous case, the intensity for such pixels is the difference between the intensity contributions of the two edges. $I = F_e(y1-y) - F_e(y2-y)$.

6. One corner of the trapezoid intersects the pixel (Figure 6-13).

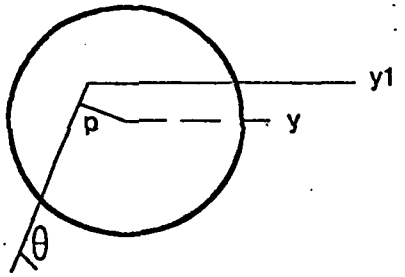


Figure 6-13:

To determine the intensity of the pixel in this case we need the distances to both the edges and the angle subtended between them. This can be formulated as $I = F_t(y-y1,p,\theta)$.

7. In the case when the trapezoid is a triangle, a corner of the triangle intersects the pixel (Figure 6-14). This case can be considered as the difference of two corners. Hence $I = F_t(y-y1,p1,\theta 1) - F_t(y-y1,p2,\theta 2)$.

8. Two corners which are connected by a horizontal edge intersect the pixel (Figure 6-15). This situation is similar to the previous one. $I = F_t(y-y1,p1,\theta 1) - F_t(y-y1,p2,\theta 2)$.

9. Two corners which are connected by a side edge intersect the pixel (Figure 6-16). This can also be formulated as the difference of two corners. $I = F_t(y-y1,p,\theta) - F_t(y-y2,p,\theta)$.

10. Three corners of the trapezoid intersect the pixel (Figure 6-17). The intensity of the pixel can be determined as the difference between three corners. $I = F_t(y-y1,p1,\theta 1) - F_t(y-y1,p2,\theta 2) - F_t(y-y2,p1,\theta 1)$.

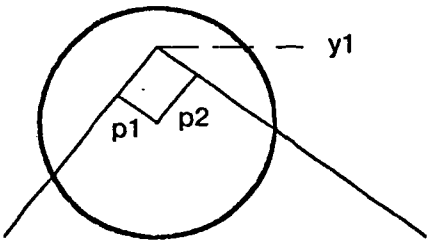


Figure 6-14:

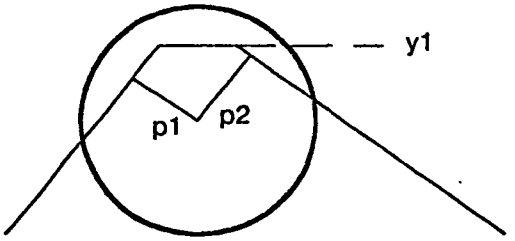


Figure 6-15:

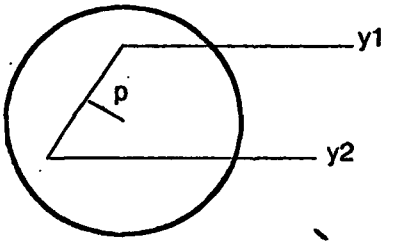


Figure 6-16:

11. The trapezoid is completely enclosed in the pixel (Figure 6-18). This situation is similar to the previous one. $I = F_t(y-y1,p1,\theta1) - F_t(y-y1,p2,\theta2) - F_t(y-y2,p1,\theta1)$.

Because F_e is a subset of F_t with $y = 1.0$, compiling a table for F_t is sufficient to determine the intensity in all possible situations. Appendix A contains this table for the conical filter with $r = 1$. As seen in the previous tables, the distance parameters of the table span the range between $-12/16$ and $+12/16$ at a spacing of $1/16$. The angular parameter has the range between 0 and $\pi/2$ with a spacing of $\pi/8$. The reason for the angular spacing is easily seen by the fact that we are using a circular filter

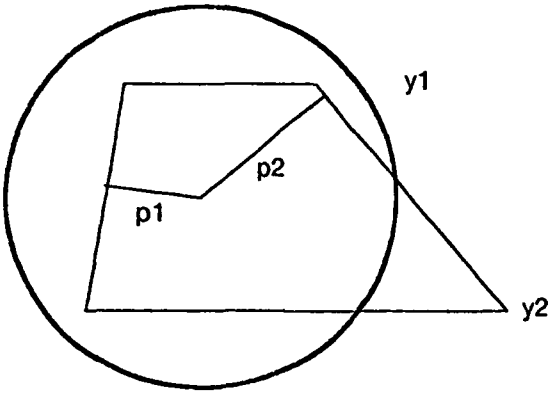


Figure 6-17:

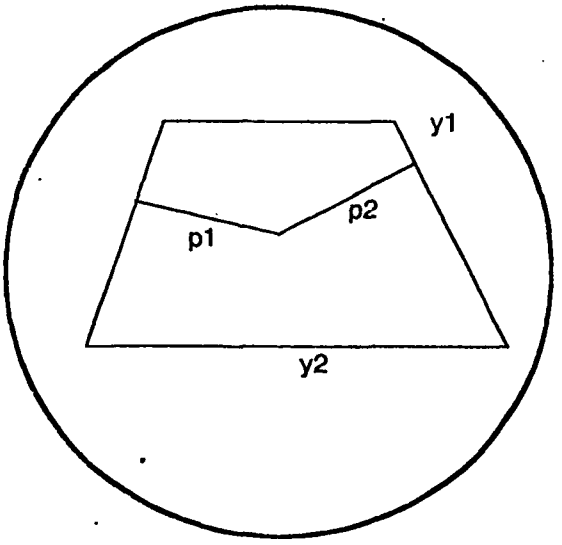


Figure 6-18:

and a 4 bit gray-scale resolution. The total angle of 2π subtended at the center of the pixel is divided by the total number of gray-levels to give the angular spacing of $2\pi/16$, i.e. $\pi/8$.

The reason for discussing the various intersections of the trapezoid with the pixel was that we can save effort if we can recognize that the intersection is a simple case as opposed to assuming that all the edges intersect the pixel. If the algorithm used does not identify the simpler cases, then we can

always assume the most complicated case, and compute the vertical distances $y1$ and $y2$ to the horizontal edges and the perpendicular distances $p1$ and $p2$ to the side edges. These distances are then thresholded to $+12/16$ or $-12/16$, and the thresholded values used to lookup the intensity table. The angles subtended by the side edges $\theta 1$ and $\theta 2$ remain constant throughtout the whole computation and can be used to set up pointers to the appropriate two-dimensional tables which then require only the y and p indices.

The next chapter discusses algorithms used to fill trapezoids using the universal table.

6.2. Combining filtered images

The previous section discussed a method for anti-aliasing images filtered by computing the images to a higher resolution and then averaging several neighboring pixels to compute the filtered intensities. We also presented a computationally cheaper method which filters the images by using geometrical information to compute the intensities for individual pixels directly.

When the only information retained is the intensity values of individual pixels, we face a problem for filtering two or more interacting lines or edges properly. The problem is faced when we compute the second of two interacting geometries because the pixels computed may already have an intensity value stored in them with no hint about the geometry of the edge that passed through them. One possible situation is shown in Figure 6-19, where the amount of the overlapping area between the two edges would have to be known in order to compute the correct intensity of the combined geometry. There is absolutely no way to deduce the overlapping area if the only information retained from the edge computed first is the intensity value for the pixel.

Some simplifying assumptions can allow us to approximate this combination process. One such assumption is often valid when the objects are part of a three-dimensional scene. The individual objects are then computed in a deepest-object-first order. Figure 6-20 shows two such objects which intersect the same pixel. The first object is deeper and hence the intensity $I1$ was computed before $I2$. The assumption we shall make is that the two objects are relatively far apart so that the light passed through the one object is uniformly dispersed by the time it gets to the second object. We shall also assume that $I1$ and $I2$ are fractional values and 1 is the maximum intensity.

For a dark background, the total light reflected has an $I2$ contribution from the second object and a contribution of $(1 - I2)I1$ from the first object. The resultant intensity is hence

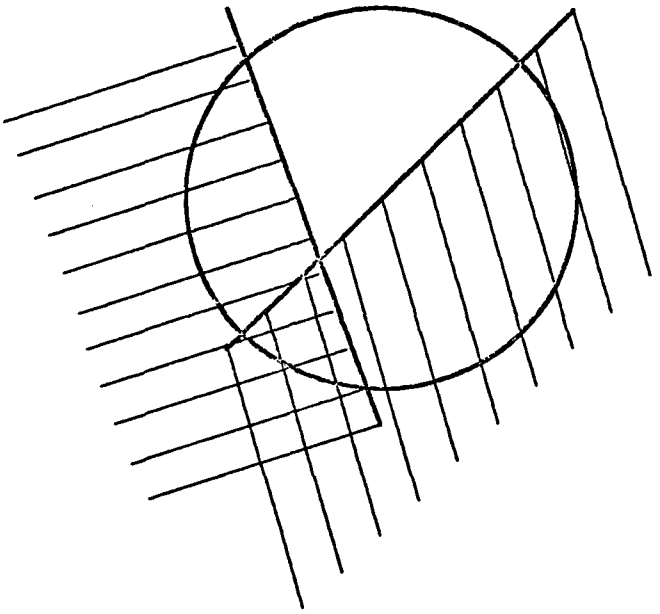


Figure 6-19: Interacting geometries within a pixel.

$$I = I2 + (1-I2)I1$$
$$= I1 + I2 - I1\,I2$$

This formulation is equivalent to one which merely multiplies the black areas of each computation.

$$I = 1 - (1-I1)(1-I2)$$
$$= I1 + I2 - I1\,I2$$

For white backgrounds, the deeper object transmits an intensity $(1-I1)$ which is diminished to $(1-I1)(1-I2)$ by the second object. The resultant intensity for white backgrounds is hence

$$I = (1-I1)(1-I2)$$
$$= 1 - I1 - I2 + I1\,I2.$$

The $I1+I2-I1\,I2$ formulation performs the correct reasonable approximation under the appropriate situations. When both $I1$ and $I2$ are small fractions, indicating that only small portions of the pixel are intersected by the objects, then the combination function will result in approximately the sum of the two intensities. This approximation is fairly reasonable because the probability of the two small portions overlapping each other is very small. In the case when one of the intensities is much larger than the other one, then the combination of the two is nearly equal to the larger one, which corresponds to the most likely situation of the larger object completely overlapping the smaller object.

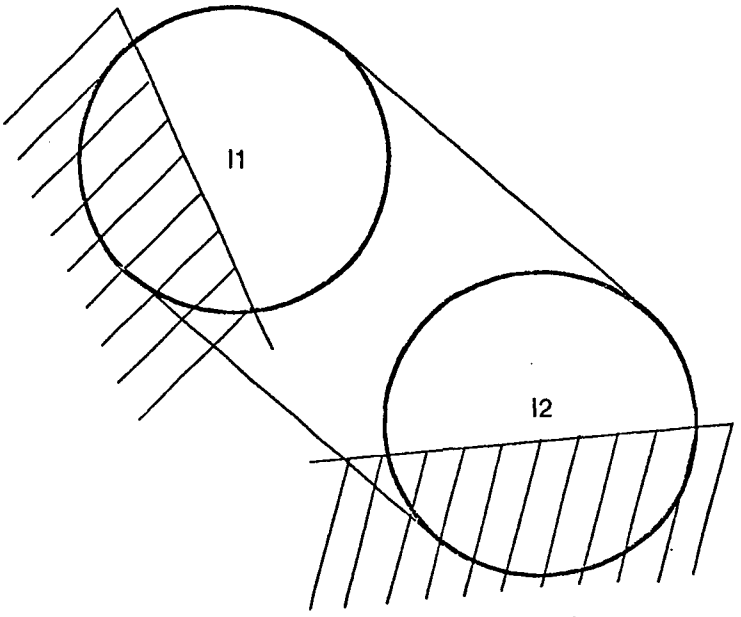


Figure 6-20: Overlapping objects within a pixel.

Several simpler approximations are often used to combine the intensity values computed by filtering image objects. The popular ones are:

$I = \text{Max}(I1, I2)$ for images with a dark background,
$I = \text{Min}(I1, I2)$ for images with a white background,
$I = I1 + I2$ assumes that the objects do not overlap

It should be noted that under known circumstances one of the easier combination functions might be used. For example when it is known *a priori* that the objects abut each other, the intensities must simply be added to compute the resultant intensity. This situation occurs when polygons are created by combining trapezoids.

Another solution to the combination problem is to keep a few bits of geometry information with each intensity value. This information can come from the tables used to look up the intensities for particular geometries and can be used to correct intensities for pixels where different edges interact. If the final image has a 4-bit resolution, then the area of the pixel should be divided into 16 equal parts, and 16 extra bits per pixel would keep track if the corresponding part of the pixel is covered or not. Since we are using a circular filter, the pixel would be divided radially (Figure 6-21). The combination function would then be

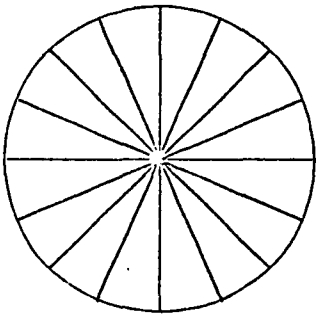


Figure 6-21: The pixel area can be divided radially to track which part of the pixel is already covered.

$$I = I1 + I2 - BitsSet(G1 \text{ AND } G2)/16,$$

where $G1$ and $G2$ are the 16-bit numbers corresponding to the geometry information in the old and the new intensities. The function *BitsSet* counts the number of bits set in its argument. This approach effectively takes us back to computing the image to a higher virtual resolution in which the combination problem does not exist to start with.

Experiments with lines and polygons conclude that the combination approximations introduced in this section produce satisfactory results for static images. It is easy to deal with known geometrical situations when the correct solution can be used. An example is the case of butting line segments and trapezoids, when adding the intensities produces the correct result. Unknown geometrical situations call for approximate combination functions. The *Max* combination function works satisfactorily for intersecting lines and edges with dark backgrounds.

The next chapter shall introduce various scan-conversion algorithms that can be used to create some of the popular geometries that use the filtering techniques discussed in this chapter.

6.3. Summary

This chapter has presented efficient filtering techniques that can be used to smoothen the edges of polygons and lines. The technique primarily uses table lookup as a means of parametrizing the computations with respect to the filter function. It should be emphasized again that the filter function should be parameter of the computations. Both the size and the shape of the filter function are usually varied until a satisfactory image results. The changing of these parameters reflects as a change

in the tables as well as the algorithms and the precision used in the algorithms using these tables. This chapter does not address the problem of how the filter function affects the algorithmic aspect of the computation.

The chapter also studies the problems of combining the filtered intensities of different objects. The combination is done in the intensity domain, which only represents a logarithmic value of the actual photon energy. The intensity values are translated into energy in a table lookup step in the display, a step usually referred to as gamma correction. The combination function is an important step in obtaining good images, and is usually also varied.

Appendix A

This appendix tabulates the function $F_t(y,p,\theta)$. Each row of the table contains the values for $-12/16 \le p \le 12/16$ at intervals for 1/16, for given values of y and θ. The values of y lie in the range $-12/16 \le y \le 12/16$ at intervals for 1/16. θ is in the range of $0 \le \theta \le \pi/2$ at intervals of $\pi/8$; the intensities for other angles can easily be derived.

$y = -12/16, \theta = 0$
0 0 1 1 1 2 3 3 4 5 6 7 8 9 10 10 11 12 13 13 14 14 15 15 15
$y = -12/16, \theta = \pi/4$
0 0 1 1 2 2 3 3 4 5 6 7 8 9 10 10 11 12 13 13 14 14 15 15 15
$y = -12/16, \theta = \pi/2$
0 1 1 1 2 2 3 4 4 5 6 7 8 9 10 11 11 12 13 13 14 14 15 15 15
$y = -11/16, \theta = \pi/8$
0 0 0 1 1 2 2 3 4 5 6 6 7 8 9 10 11 12 12 13 14 14 15 15 15
$y = -11/16, \theta = 3\pi/8$
0 1 1 1 2 2 3 3 4 5 6 7 8 8 9 10 11 12 12 13 14 14 15 15 15
$y = -10/16, \theta = 0$
0 0 0 0 1 1 2 3 4 4 5 6 7 8 9 10 11 11 11 12 13 13 14 14 15 15
$y = -10/16, \theta = \pi/4$
0 0 1 1 1 2 2 3 4 4 5 6 7 8 9 10 11 11 11 12 13 13 14 14 15 15
$y = -10/16, \theta = \pi/2$
0 1 1 1 2 2 3 3 4 5 6 7 8 8 9 10 11 12 12 12 13 13 14 14 15 15
$y = -9/16, \theta = \pi/8$
0 0 0 0 1 1 2 2 3 4 5 6 7 8 9 9 10 11 12 12 13 13 14 14 14
$y = -9/16, \theta = 3\pi/8$
0 0 1 1 1 2 2 3 4 4 5 6 7 8 9 10 10 11 12 12 13 13 14 14 14
$y = -8/16, \theta = 0$
0 0 0 0 0 1 1 2 3 4 4 5 6 7 8 9 10 11 11 11 12 12 13 13 14 14
$y = -8/16, \theta = \pi/4$
0 0 0 1 1 1 2 2 3 4 5 5 6 7 8 9 10 11 11 11 12 12 13 13 14 14
$y = -8/16, \theta = \pi/2$
0 1 1 1 2 2 3 3 4 5 5 6 7 8 9 10 10 11 12 12 13 13 13 14 14
$y = -7/16, \theta = \pi/8$
0 0 0 0 0 1 1 2 2 3 4 5 6 7 8 8 9 10 11 11 11 12 12 13 13 13
$y = -7/16, \theta = 3\pi/8$
0 0 1 1 1 2 2 3 3 4 5 5 6 7 8 9 9 10 11 11 11 12 12 13 13 13
$y = -6/16, \theta = 0$
0 0 0 0 0 0 0 1 1 2 3 4 5 6 7 8 9 9 10 11 11 11 12 12 12 13
$y = -6/16, \theta = \pi/4$
0 0 0 0 1 1 1 2 2 3 4 4 5 6 7 8 9 9 10 11 11 11 12 12 12 13

$y = -12/16, \theta = \pi/8$
0 0 1 1 1 2 3 3 4 5 6 7 8 9 10 10 11 12 13 13 14 14 15 15 15
$y = -12/16, \theta = 3\pi/8$
0 1 1 1 2 2 3 3 4 5 6 7 8 9 10 10 11 12 13 13 14 14 15 15 15
$y = -11/16, \theta = 0$
0 0 0 1 1 2 2 3 4 5 6 6 7 8 9 10 11 12 13 13 14 14 15 15 15
$y = -11/16, \theta = \pi/4$
0 0 1 1 1 2 2 3 4 5 6 6 7 8 9 10 11 12 12 13 14 14 15 15 15
$y = -11/16, \theta = \pi/2$
0 1 1 1 2 2 3 4 4 5 6 7 8 9 9 10 11 12 13 13 14 14 15 15 15
$y = -10/16, \theta = \pi/8$
0 0 0 1 1 1 2 3 4 4 5 6 7 8 9 10 11 11 12 13 13 14 14 15 15
$y = -10/16, \theta = 3\pi/8$
0 1 1 1 2 2 3 3 4 5 6 6 7 8 9 10 11 11 12 13 13 14 14 15 15
$y = -9/16, \theta = 0$
0 0 0 0 0 1 2 2 3 4 5 6 7 8 9 9 10 11 12 12 13 13 14 14 14
$y = -9/16, \theta = \pi/4$
0 0 0 1 1 2 2 2 3 3 4 5 6 7 8 9 9 10 11 12 12 13 13 14 14 14
$y = -9/16, \theta = \pi/2$
0 1 1 1 2 2 3 3 4 5 6 6 7 8 9 10 11 11 12 13 13 13 14 14 14
$y = -8/16, \theta = \pi/8$
0 0 0 0 0 1 1 2 3 3 4 5 6 7 8 9 10 11 11 12 12 13 13 14 14
$y = -8/16, \theta = 3\pi/8$
0 0 1 1 1 2 2 3 4 4 5 6 7 7 8 9 10 11 11 12 12 13 13 14 14
$y = -7/16, \theta = 0$
0 0 0 0 0 0 1 1 2 3 4 5 6 7 8 8 9 10 11 11 12 12 13 13 13
$y = -7/16, \theta = \pi/4$
0 0 0 1 1 1 2 2 3 3 4 5 6 7 8 8 9 10 11 11 12 12 13 13 13
$y = -7/16, \theta = \pi/2$
0 1 1 1 2 2 3 3 4 5 5 6 7 8 8 9 10 11 11 12 12 13 13 13 13
$y = -6/16, \theta = \pi/8$
0 0 0 0 0 0 1 1 2 2 3 4 5 6 7 8 9 9 10 11 11 11 12 12 12 13
$y = -6/16, \theta = 3\pi/8$
0 0 1 1 1 2 2 2 3 4 4 5 6 7 7 8 9 10 10 11 11 12 12 12 13

$y = -6/16, \theta = \pi/2$
0 1 1 1 2 2 2 3 4 4 5 6 7 7 8 9 9 10 11 11 12 12 12 13 13
$y = -5/16, \theta = \pi/8$
0 0 0 0 0 0 0 1 1 2 3 3 4 5 6 7 8 9 9 10 11 11 11 11 12 12
$y = -5/16, \theta = 3\pi/8$
0 0 0 1 1 1 2 2 2 3 3 4 5 5 6 7 8 8 9 10 10 11 11 11 11 12 12
$y = -4/16, \theta = 0$
0 0 0 0 0 0 0 0 0 1 2 3 4 5 5 6 7 8 9 9 10 10 11 11 11
$y = -4/16, \theta = \pi/4$
0 0 0 0 0 1 1 1 2 2 3 3 4 5 6 7 7 8 9 9 10 10 11 11 11
$y = -4/16, \theta = \pi/2$
0 0 1 1 1 2 2 2 3 3 4 4 5 6 6 7 8 8 9 9 10 10 11 11 11 11
$y = -3/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 1 1 1 2 3 4 5 6 6 7 8 8 9 9 10 10 10
$y = -3/16, \theta = 3\pi/8$
0 0 0 1 1 1 1 2 2 2 3 3 4 4 5 6 6 7 8 8 9 9 10 10 10 10
$y = -2/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 5 6 7 8 8 9 9 9 10
$y = -2/16, \theta = \pi/4$
0 0 0 0 0 0 1 1 1 1 2 2 3 4 4 5 6 6 7 8 8 9 9 9 10
$y = -2/16, \theta = \pi/2$
0 0 1 1 1 1 2 2 2 3 3 4 4 5 6 6 6 7 7 8 8 8 9 9 9 10
$y = -1/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 4 5 5 6 7 7 8 8 8 9
$y = -1/16, \theta = 3\pi/8$
0 0 0 0 1 1 1 1 2 2 2 2 3 3 4 5 5 5 6 6 7 7 7 8 8 9
$y = 0/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 4 5 6 6 7 7 7 8
$y = 0/16, \theta = \pi/4$
0 0 0 0 0 0 0 0 1 1 1 2 2 2 3 3 4 4 5 5 6 6 7 7 8
$y = 0/16, \theta = \pi/2$
0 0 0 1 1 1 1 2 2 2 3 3 4 4 4 5 5 5 6 6 7 7 7 8 8
$y = 1/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 2 3 4 4 5 5 5 6 6 6 7
$y = 1/16, \theta = 3\pi/8$
0 0 0 0 0 0 1 1 1 1 2 2 2 3 3 3 4 4 5 5 5 5 6 6 6 7 7
$y = 2/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 3 4 4 5 5 5 6 6
$y = 2/16, \theta = \pi/4$
0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 2 2 3 3 4 4 5 5 5 6 6
$y = 2/16, \theta = \pi/2$
0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 4 4 4 5 5 5 5 6 6 6 6
$y = 3/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 2 3 3 4 4 5 5
$y = 3/16, \theta = 3\pi/8$
0 0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 4 4 4 4 5 5 5
$y = 4/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 3 3 4 4 4
$y = 4/16, \theta = \pi/4$
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 3 3 3 4 4 4
$y = 4/16, \theta = \pi/2$
0 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 4 4 4
$y = 5/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 2 3 3 3
$y = 5/16, \theta = 3\pi/8$
0 0 0 0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3
$y = 6/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 2 3
$y = 6/16, \theta = \pi/4$
$y = -5/16, \theta = 0$
0 0 0 0 0 0 0 0 1 2 3 3 4 5 6 7 8 9 9 10 11 11 11 11 12 12
$y = -5/16, \theta = \pi/4$
0 0 0 0 1 1 1 2 2 2 3 3 4 5 6 6 7 8 9 9 10 11 11 11 11 12 12
$y = -5/16, \theta = \pi/2$
0 0 1 1 1 1 2 2 2 3 3 4 5 5 5 6 7 8 8 8 9 9 10 11 11 11 11 12 12 12
$y = -4/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 1 1 2 3 4 5 5 6 7 8 9 9 10 10 11 11 11
$y = -4/16, \theta = 3\pi/8$
0 0 0 1 1 1 1 2 2 2 3 4 4 5 6 6 7 8 8 9 9 10 10 11 11 11
$y = -3/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 6 6 7 8 8 9 9 10 10 10
$y = -3/16, \theta = \pi/4$
0 0 0 0 0 0 1 1 1 2 2 3 4 4 5 6 7 7 8 8 9 9 10 10 10
$y = -3/16, \theta = \pi/2$
0 0 1 1 1 1 2 2 2 3 4 4 5 5 6 7 7 8 8 9 9 10 10 10 10 11
$y = -2/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 1 1 2 2 3 4 5 5 6 7 8 8 9 9 10
$y = -2/16, \theta = 3\pi/8$
0 0 0 0 1 1 1 2 2 2 3 3 4 5 5 6 6 7 7 8 8 9 9 10
$y = -1/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 4 5 5 6 7 7 8 8 8 9
$y = -1/16, \theta = \pi/4$
0 0 0 0 0 0 0 1 1 1 2 2 3 3 4 4 5 6 6 7 7 8 8 8 9
$y = -1/16, \theta = \pi/2$
0 0 1 1 1 1 2 2 2 3 3 4 4 5 6 6 6 7 7 8 8 8 8 9 9
$y = 0/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 4 4 5 6 6 7 7 8
$y = 0/16, \theta = 3\pi/8$
0 0 0 0 0 1 1 1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 8 8
$y = 1/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 3 3 4 5 5 6 6 6 7
$y = 1/16, \theta = \pi/4$
0 0 0 0 0 0 0 0 1 1 1 2 2 2 3 3 4 4 5 5 6 6 6 7
$y = 1/16, \theta = \pi/2$
0 0 0 1 1 1 1 2 2 2 3 3 4 4 4 5 5 5 6 6 6 6 7 7 7
$y = 2/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 3 3 4 4 5 5 6 6
$y = 2/16, \theta = 3\pi/8$
0 0 0 0 0 0 0 1 1 1 1 2 2 2 3 3 4 4 4 5 5 5 6 6
$y = 3/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 2 2 3 4 4 4 5 5
$y = 3/16, \theta = \pi/4$
0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 3 3 4 4 4 5 5
$y = 3/16, \theta = \pi/2$
0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5
$y = 4/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 3 3 4 4 4
$y = 4/16, \theta = 3\pi/8$
0 0 0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 4 4 4 4
$y = 5/16, \theta = 0$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 2 2 3 3 3
$y = 5/16, \theta = \pi/4$
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3
$y = 5/16, \theta = \pi/2$
0 0 0 0 0 0 1 1 1 1 1 2 2 2 2 2 3 3 3 3 3 3 4 4
$y = 6/16, \theta = \pi/8$
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 2 2 2 3
$y = 6/16, \theta = 3\pi/8$

0000000000000011111222223
$y = 6/16, \theta = \pi/2$
0000000111111222222333333
$y = 7/16, \theta = \pi/8$
0000000000000000000111122
$y = 7/16, \theta = 3\pi/8$
0000000000001111112222222
$y = 8/16, \theta = 0$
0000000000000000000000111
$y = 8/16, \theta = \pi/4$
0000000000000000011111112
$y = 8/16, \theta = \pi/2$
0000000001111111112222222
$y = 9/16, \theta = \pi/8$
0000000000000000000000111
$y = 9/16, \theta = 3\pi/8$
0000000000000001111111111
$y = 10/16, \theta = 0$
0000000000000000000000001
$y = 10/16, \theta = \pi/4$
0000000000000000000000111
$y = 10/16, \theta = \pi/2$
0000000000000111111111111
$y = 11/16, \theta = \pi/8$
0000000000000000000000000
$y = 11/16, \theta = 3\pi/8$
0000000000000000000000111
$y = 12/16, \theta = 0$
0000000000000000000000000
$y = 12/16, \theta = \pi/4$
0000000000000000000000000
$y = 12/16, \theta = \pi/2$
0000000000000000000000000

0000000000011111222222333
$y = 7/16, \theta = 0$
0000000000000000000011122
$y = 7/16, \theta = \pi/4$
0000000000000000111112222
$y = 7/16, \theta = \pi/2$
0000000011111112222222222
$y = 8/16, \theta = \pi/8$
0000000000000000000001111
$y = 8/16, \theta = 3\pi/8$
0000000000000111111111222
$y = 9/16, \theta = 0$
0000000000000000000000011
$y = 9/16, \theta = \pi/4$
0000000000000000000111111
$y = 9/16, \theta = \pi/2$
0000000000011111111111111
$y = 10/16, \theta = \pi/8$
0000000000000000000000001
$y = 10/16, \theta = 3\pi/8$
0000000000000000000111111
$y = 11/16, \theta = 0$
0000000000000000000000000
$y = 11/16, \theta = \pi/4$
0000000000000000000000000
$y = 11/16, \theta = \pi/2$
0000000000000000000111111
$y = 12/16, \theta = \pi/8$
0000000000000000000000000
$y = 12/16, \theta = 3\pi/8$
0000000000000000000000000

Chapter 7
Filtered Scan-Conversion

This chapter shows that algorithms for scan-conversion with filtering are similar to the algorithms for bit-map scan-conversion discussed before with the only difference being the need for extra precision in distance parameters to allow the computation of gray-scale intensities. The amount of extra precision required is approximately the same as the number of bits of precision in the pixel intensity values. In this chapter we shall restrict our discussion to unit-thickness lines and trapezoids, and the use of the conical filter with unit radius.

7.1. Filtered line drawing

The first subsection shall present a simple extension to the Bresenham algorithm that can be used to draw gray-scale lines. The second subsection shall discuss algorithms that allow the parallel update of several pixels at a time.

7.1.1. Incremental algorithms for filtering edges

In Chapter 6 we presented a table that can be used to determine the intensities for the pixels that intersect the edges of a line. Only the perpendicular distance from the pixel to the center of the line is required to determine the intensity for that pixel. Hence the algorithms to draw a line must compute the perpendicular distance p between each pixel center and the line. To reduce computation, this calculation is performed incrementally, as in Bresenham's algorithm [Bresenham 65]. This discussion shall be restricted to unit thickness lines drawn from (0,0) to (dx,dy) in the first octant (i.e., $0 \leq dy \leq dx$); extensions to other lines are obvious. Such lines intensify two or three pixels in each column of pixels (see Figure 7-1)[5]. The algorithm will keep track of the location of the center pixel and the perpendicular distance to the line's center from the pixel center.

[5]The algorithms and tables presented in this chapter are designed to produce images using 4-bit intensity values. A diagonal line at almost 45 degrees will actually intersect five rather than three pixels, but the top and bottom pixels are intensified at less than 0.2% of the maximum. The algorithm presented ignores these pixels because a 4-bit intensity value will record zero for such an intensity. If a wider range of intensities is available, then algorithm may be modified in an obvious way to illuminate more pixels in each column; the tables must also provide more precision.

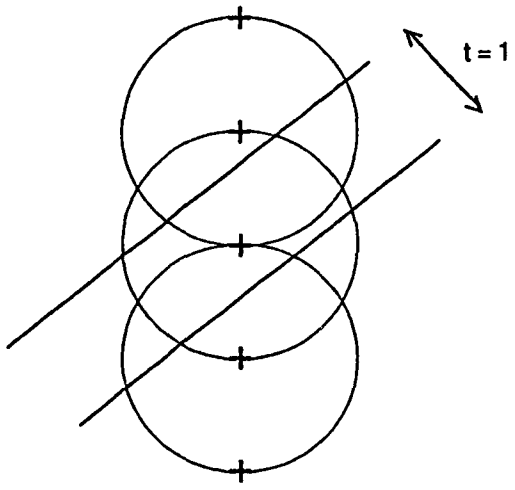


Figure 7-1: Three pixels are shaded in each column.

We shall initially assume that line endpoints lie at pixel centers, and hence dx and dy are integers. From our first-octant assumption, the slope $m = dy/dx$ has a value between 0 and 1. In the algorithm below, x and y track the central pixel in each column through which the line passes, and v is the vertical distance from that pixel to the line. This vertical distance v is a signed value; a positive value indicates that the center of the line is above the center of the pixel[6], and a negative value indicates the opposite. The variable s is a threshold distance used to decide whether the central pixel in the next column lies diagonally or horizontally across from the central pixel in the current column.

```
VAR x,y, : INTEGER; v,m,s : REAL;
m := dy/dx;
v := 0; s := 0.5-m;
y := 0;
FOR x := 0 to dx DO
BEGIN
    Shade pixels at (x,y-1), (x,y) (x,y+1);
    IF (v ≥ s) THEN
        BEGIN
            y := y+1;
            v := v+m-1;
        END
    ELSE
        v := v+m;
END;
```

The pixel at (x,y) is located at a vertical distance v from the line, and the pixels at $(x,y-1)$ and $(x,y+1)$ are at distances $v-1$ and $v+1$ respectively.

---

[6] y increases upwards

In order to determine the shade of the pixels, we need to compute the perpendicular distance p from the pixel to the line. The vertical distances are related to the perpendicular distances by a factor of $c = dx/\mathrm{sqrt}(dx^2+dy^2)$, such that $p = cv$. The following algorithm shows the modifications to compute the perpendicular distance.

```
VAR x,y, : INTEGER; p,m,c,s : REAL;
m := dy/dx;
c := 1/sqrt(m*m+1);
p := 0; s := (0.5-m)*c;
y := 0;
FOR x := 0 to dx DO
BEGIN
    Shade pixels at (x,y-1), (x,y) (x,y+1);
    IF (p ≥ s) THEN
        BEGIN
            y := y+1;
            p := p+(m-1)*c;
        END
    ELSE
        p := p+m*c;
END;
```

The two expressions $(m-1)c$ and mc can be precomputed and do not have to be computed repeatedly in the inner loop. To compute the pixel shades, the absolute values of p, $p-c$, and $p+c$ are used as indices into Table 6-2 to determine intensities at (x,y), $(x,y-1)$, and $(x,y+1)$ respectively. The table look-up is performed by thresholding the absolute distances to 20/16 and using the new distance value to index into the table.

The incremental algorithm presented above does not compute the sampled values of pixels on or near the endpoints of the line. The situation is illustrated in Figure 7-2, which shows an endpoint of a line. The pixels shown with their surrounding filters are not intensified correctly by our algorithm; in fact some of them are not intensified at all. The intensities of each of these pixels is obtained by integrating the filter function for each pixel with the exact shape of the line. There are several methods available to perform such computations.

One approach to computing the endpoint intensities is by using exact geometric operations [Feibush 80] to split the endpoint into geometries for which the sampled intensity is precomputed. An approximation can be obtained by sampling the endpoint at points much more closely spaced than pixel centers. Both these approaches are computationally expensive.

An easier approach is to precompute the endpoint intensities and store them in a table, much as we do for line intensities. To display the endpoint of a line, the intensities of the six pixels in the vicinity are determined from the table. For 4-bit gray-scale accuracy, it is sufficient to compute a set of

endpoints for lines with slopes between 0 and 1 at intervals of 1/16 in slope (see Table 7-1). The entries in the table are in the same configuration as the six pixels shown in Figure 7-2. Using a combination of mirroring and transposition transformations, these endpoint intensities can be used for lines in every octant.

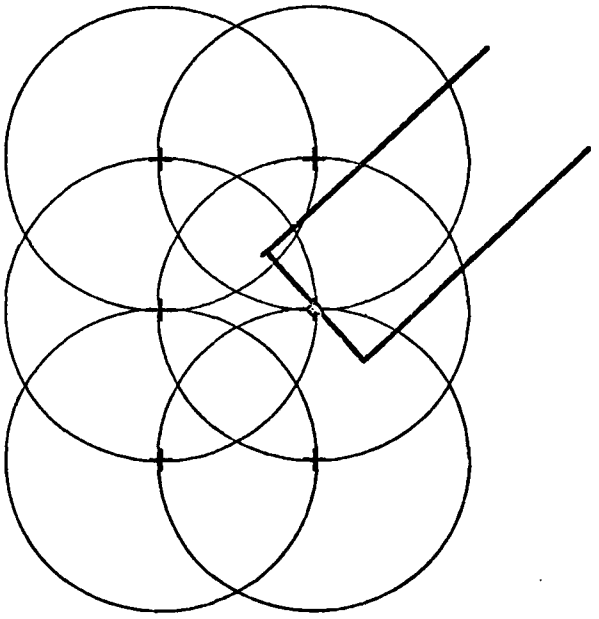


Figure 7-2: End of a line, showing the six pixels that may be illuminated.

The table look-up technique for endpoint intensities allows us to alter the anti-aliasing filter function and the endpoint shape by simply replacing the table. As discussed before, the filter function is varied until the resulting image is aesthetically satisfactory. The endpoints of lines can also have different shapes; the flat and circular ones are normally used.

The algorithm can be adapted to display lines and edges drawn between endpoints that are not integers; that is, that do not lie on the pixel grid. Such a scheme is necessary to avoid additional aliasing, for example the jitter in a moving image caused by quantizing endpoints to lie on pixel centers. To accommodate non-integer endpoints, two modifications must be made. First, the initialization of the algorithm must be changed to compute the coordinate of the first pixel to be illuminated and to compute the initial value for p at this point. The following algorithm shows these modifications. The non-integral endpoints are (x_1,y_1) and (x_2,y_2).

Slope = 0/16		Slope = 1/16		Slope = 2/16	
0.000	0.056	0.000	0.063	0.000	0.072
0.000	0.393	0.000	0.390	0.000	0.390
0.000	0.056	0.000	0.048	0.000	0.042
Slope = 3/16		**Slope = 4/16**		**Slope = 5/16**	
0.000	0.082	0.000	0.094	0.000	0.107
0.000	0.390	0.000	0.390	0.000	0.390
0.000	0.037	0.000	0.032	0.000	0.028
Slope = 6/16		**Slope = 7/16**		**Slope = 8/16**	
0.000	0.121	0.000	0.136	0.000	0.152
0.001	0.390	0.001	0.390	0.002	0.391
0.000	0.025	0.000	0.022	0.000	0.019
Slope = 9/16		**Slope = 10/16**		**Slope = 11/16**	
0.000	0.169	0.000	0.187	0.000	0.206
0.002	0.390	0.003	0.390	0.003	0.390
0.000	0.017	0.000	0.015	0.000	0.013
Slope = 12/16		**Slope = 13/16**		**Slope = 14/16**	
0.000	0.225	0.000	0.245	0.000	0.264
0.004	0.390	0.005	0.390	0.006	0.390
0.000	0.012	0.000	0.010	0.000	0.009
Slope = 15/16		**Slope = 16/16**			
0.001	0.284	0.001	0.304		
0.007	0.390	0.007	0.391		
0.000	0.008	0.000	0.007		

Table 7-1: Endpoints for lines with different slopes.

```
VAR x,y : INTEGER; p,m,c,s : REAL;
m := (y2-y1)/(x2-x1);
c := 1/sqrt(m*m+1);
y1 := y1+m*(round(x1)-x1);
y := round(y1);
p := (y1-y)*c ;
s := (0.5-m)*c;
FOR x := round(x1) TO round(x2) DO
BEGIN
  Shade pixels at (x,y-1),(x,y),(x,y+1);
  IF (p ≥ s) THEN
    BEGIN
      y := y+1;
      p := p+(m-1)*c;
    END
  ELSE
    p := p+m*c;
END;
```

The second change is that samples near endpoints need to take account of the exact location of the line endpoint. We either have to spend a lot of processing to filter these pixels or use a much larger table to look up the filtered values. If the table contains endpoints at intervals of 1/16 for each coordinate and an interval of 1/16 for the slope, then the size of the table at six pixels per endpoint would be 29478 entries! Chapter 8 discusses an algorithm which can be used move images by subpixel distances. This would reduce the storage requirements of the endpoint tables.

7.1.2. Drawing lines using strokes

As discussed in the Chapter 5, the ability to update several pixels at a time can be used to increase the speed of line drawing algorithms. The line drawing algorithms can now compose lines using shorter segments which are joined together to form the line. This technique can be used effectively in display architectures which allow parallel update, the primary subject of this thesis. As we have seen before, line drawing requires the ability to update a square area in order to achieve a symmetric speedup for both shallow and steep lines.

We shall present two different strategies which draw lines using parallel updates. The first scheme computes each stroke by using a parallel set of processors, while the second one uses a precomputed set of strokes and the line drawing algorithm merely chooses which of the set of strokes to use and where to place them. Both techniques are described for lines with integer endpoints in the first octant originating at (0,0) and drawn to (dx,dy). They can be extended to non-integer endpoints in the same manner as the incremental algorithm.

7.1.2.1. Computed strokes

A processor-per-pixel has always been the dream of all people who design raster scan displays. If each point on the display had an independent processor, then the line drawing algorithm can use this immense parallelism available to draw each line in one step. Each processor would merely have to determine whether the pixel it is responsible for lies on or outside the line, and intensify the pixel appropriately. Although this technique updates the line in a time which is independent of the length of the line, it is wasteful of the processing power available because most of the processors represent pixels nowhere near the line. A tradeoff would have a set of processors which can update a part of the display and use them iteratively to compute the whole line. Assigning these processors to pixels is equivalent to the memory organization problem. If they can be used to update several points along a scan in parallel, then horizontal lines can be drawn faster than vertical lines and vice versa. For this reason these processors should update a square region of the display.

If the set of processors can be used to update a NxN square region of the display, then lines are drawn by updating N-pixel strokes at a time. The line drawing algorithm would position the processors at a certain point on the display, compute the intensities of all pixels in the square and then update the memory. The processors would then be positioned to the next point along the line and the steps iterated until the whole line is drawn.

Once again we shall restrict the discussion to lines drawn from (0,0) to (dx,dy) in the first octant. The algorithm described can be modified easily for other lines in other octants. As seen before a gray-scale line in the first octant with unit thickness illuminates two or three pixels in each column. Hence a diagonal line illuminates pixels in $N+1$ rows for a span of N columns (see Figure 7-3). Since the display only allows the update of NxN squares, our algorithm can step by only $N-1$ column for each update. Conversely, lines in the second octant will be drawn by stepping up $N-1$ rows.

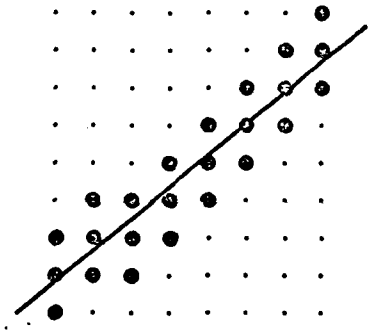


Figure 7-3: $N+1$ rows of pixels are illuminated in a span of N columns.

Our line drawing algorithm makes two assumptions. First, we assume that all pixels turned on are within the rectangle enclosed by the two end points of the line. Second, the intensity of all the pixels in this rectangle can be computed as if the line extended infinitely in both directions. This assumption produces incorrect endpoint intensities and can be fixed by placing endpoint strokes (Table 7-1) at each endpoint of the line.

The first stroke will be located at $(0,-1)$ and subsequent ones will be located at $((N-1)i, \lfloor i(N-1)dy/dx \rfloor - 1)$ for $0 \leq i \leq \lfloor dx/(N-1) \rfloor$.

Within each stroke, each pixel needs to compute the perpendicular distance to the line in order to determine the intensity of that pixel. Figure 7-4 shows this situation and demonstrates that if we have three precomputed values of

$$sin\theta = dy/\sqrt{dx^2+dy^2},$$

$$cos\theta = dx/\sqrt{dx^2+dy^2},$$

and $derr_i$ which is the perpendicular distance from the stroke origin (x_i,y_i) to the line, then each pixel (x,y) in the stroke can compute its perpendicular distance as

$$d_{xy} = (y-y_i)cos\theta - (x-x_i)sin\theta + derr_i$$

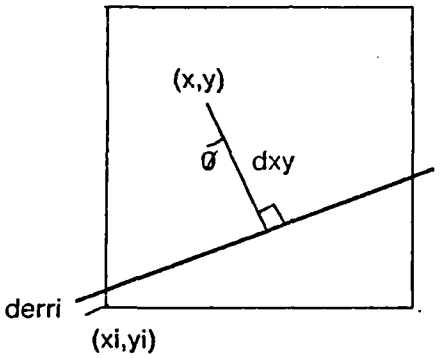


Figure 7-4: Computing the perpendicular distance to the line in parallel.

The three precomputed values can be broadcast to the NxN processor array before the distance computation. Notice that $sin\theta$ and $cos\theta$ remain constant for the whole line while $derr_i$ has to be updated for each stroke and can be computed as

$$derr_i = y_i cos\theta - x_i sin\theta.$$

The following algorithm combines these steps and uses the precomputed values of $sin\theta$, $cos\theta$, s $(=\lfloor (N-1)dy/dx \rfloor)$, and a $(=2(N-1)dx - 2s\,dx)$.

```
y_i := -1;
derr_i := -cosθ;
r := a - dx;
for x_i := 0 to dx by (N-1) do begin

    { The following nested loop is executed in parallel by the
        NxN processors }
    for x := 0 to (N-2) do
      for y := 0 to (N-1) do begin
        d_xy := y*cosθ - x*sinθ + derr_i;
        Intensity_xy := LineTable[d_xy];
        end;

    if r ≥ 0 then begin
        y_i := y_i + s + 1;
        r := r + a - 2*dx;
        derr_i := derr_i + (s+1)*cosθ - (N-1)*sinθ;
        end;
    else begin
        y_i := y_i + s;
        r := r + a;
        derr_i := derr_i + s*cosθ - (N-1)*sinθ;
        end;
    end;
```

The integer prologue used in the bit-map stroke algorithms can be used to precompute the values of a and s, while table look up can determine values for $sin\theta$ and $cos\theta$. Notice that by using incremental techniques all distance computations have been reduced to multiplications by numbers in the range between 0 and $(N-1)$, which can be performed in hardware using an adder with (log N) inputs.

7.1.2.2. Precomputed Strokes

Drawing bit-map lines using precomputed strokes was the subject of Chapter 5. It discusses several algorithms and their tradeoffs with respect to their speed and the total number of strokes required to produce an arbitrary line. All algorithms assume the ability to place the origin of any stroke at an arbitrary pixel of the display. This implies that all strokes can be defined such that the first pixel displayed is located at the origin (0,0) and the translation of every point of the stroke can be used to position the stroke at any pixel boundary. We shall once again make this assumption and also restrict this discussion to lines in the first octant. Lines in the other octants can be drawn either by using a combination of mirroring and transposing of strokes, or by using a larger set of strokes.

In the discussion on bit-map lines, we concluded that the positioning of pixels on lines drawn using strokes has two sources of error. The first is the error caused in the placement of the stroke and the

second is error within the stroke. The most simple-minded bit-map algorithms cause an error of 1/2 in the stroke placement, and another error of 1/2 within a stroke which could be in the same direction as the stroke placement error, resulting in a total possible error of approximately one. The appearance of the lines is improved by using a larger number of strokes with the added parameter of sub-pixel placement. The line drawing algorithm then computes the stroke position to a higher precision and uses the sub-pixel offset to choose from a larger set of strokes. The number of strokes increases as the square of the sub-pixel resolution.

The situation is not very different in the case of gray-scale lines. Since gray-scale intensities are computed using sub-pixel positioning of the line, an algorithm which computes the stroke positions to the same sub-pixel resolution and also uses the high-precision value of the slope available to choose from a larger set of strokes could be used to produce gray-scale lines. In our filtering example of four bits of gray-scale and a conical filter function, distances have to be computed to 1/16th of a pixel, which can be done by adding four extra bits of precision to coordinates. In practice fewer extra bits of precision produce satisfactory results.

The algorithm used for bit-map lines can also be used here for gray-scale lines. Strokes for the first octant will originate between 0.5 and 1.5 spaced by the sub-pixel resolution, and end between 0.5 and $N-0.5$ also spaced by the same sub-pixel resolution. Let i be the number of extra bits of precision provided to compute sub-pixel positioning, and $j = 2^i$. The total number of strokes required is hence $j^2(N-1)$. We can choose to index the strokes as (m,n) where m defines the origin of the stroke and varies such that $0 \le m < j$, and n is the height of the stroke such that $0 \le n < j(N-1)$.

The following algorithm can be used to draw lines in the first octant. It assumes that the line is drawn from (0,0) to (dx,dy) and a prologue is used to compute s $(=\lfloor j(N-1)dy/dx \rfloor)$, and a $(=2j(N-1)dx - 2jsdx)$.

```
y := 0;
r := a - j*dx;
for x := 0 to dx by (N-1) do begin
    if (r ≥ 0) then begin
        Display Stroke (y%j,s+1) at (x,round(y/j)-1);
        y := y + s + 1;
        r := r + a - 2*j*dx;
        end
    else begin
        Display Stroke (y%j,s) at (x,round(y/j)-1);
        y := y + s;
        r := r + a;
        end
    end
```

In this algorithm both y and s are represented with an integer part and an i-bit fractional part. Because of the stroke structure we have defined, the strokes are placed one pixel below the integer rounded value of y (i.e. at round(y/j) − 1). The stroke would hence be placed at (x,round(y/j) − 1) and line within the stroke would originate at y/j − round(y/j) + 1. The algorithm can be modified to add a constant to the initial value of y and then use the integer part of y as the origin of the stroke.

```
y := j/2;
r := a - j*dx;
for x := 0 to dx by (N-1) do begin
    if (r ≥ 0) then begin
        Display Stroke (y%j,s+1) at (x,y/j-1);
        y := y + s + 1;
        r := r + a - 2*j*dx;
        end
    else begin
        Display Stroke (y%j,s) at (x,y/j-1);
        y := y + s;
        r := r + a;
        end
    end
```

The value of i is an aesthetic decision. For a 4-bit gray-scale display $i = 2$ seems to suffice and hence a total of 112 strokes is sufficient for such display devices.

7.2. Filtered trapezoids

The ability to fill trapezoids is sufficient to fill arbitrarily shaped convex polygons. The first subsection presents an incremental algorithm to fill trapezoids. This algorithm uses the universal anti-aliasing table discussed in the previous chapter. A second subsection shall discuss parallel updating techniques to fill trapezoids.

7.2.1. Incremental algorithms for filling trapezoids

Section 6.1.2 presents a table lookup technique that can be used to determine filtered intensities for pixels that intersect trapezoids. The intensity of every pixel inside a trapezoid can be determined usng six parameters, the distances to the four edges and the angles of the two side edges. These parameters are then used in the following manner to compute the intensity of the pixel.

$$\text{PixelIntensity}(yd_1, yd_2, pd_1, pd_2, \theta_1, \theta_2)$$

$$= F_t(yd_1, pd_1, \theta_1) - F_t(yd_1, pd_2, \theta_2) - F_t(yd_2, pd_1, \theta_1)$$

where yd_1 and yd_2 are the vertical distances to the top and bottom edges respectively, pd_1 and pd_2 are

the perpendicular distances to the left and right edges respectively, and θ_1 and θ_2 are the angles subtended by the left and right edges respectively. F_t is universal polygon table.

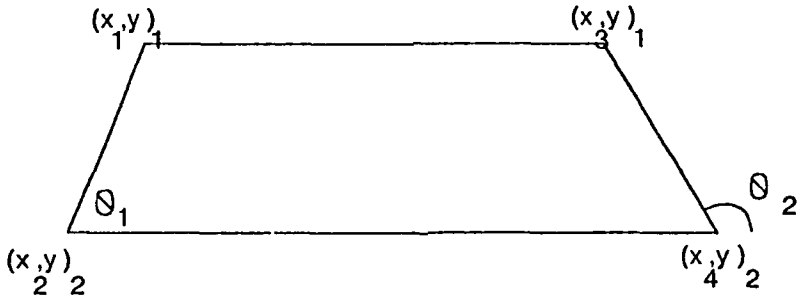


Figure 7-5: Polygon parameters.

An incremental algorithm that computes these six parameters top to bottom, left to right can be used to fill a trapezoid. The angular parameters remain constant for the whole trapezoid (Figure 7-5). This algorithm does not restrict any of the endpoints to be integers.

```
θ1 := arctan((y1-y2)/(x1-x2));
θ2 := arctan((y1-y2)/(x3-x4));
yt := round(y1)-1; yb := round(y2)+1;
xl := x1; xr := x3;
xlinc := (x2-x1)/(y2-y1);
xrinc := (x4-x3)/(y2-y1);
pdinc1 := sinθ1; pdinc2 := -sinθ2;

for y := yt to yb do begin
    yd1 := y - y1; yd2 := y2 - y;
    xl := xl + xlinc; xr := xr + xrinc;
    pd1 := (round(xl)-1-xl)*sinθ1;
    pd2 := (xr-round(xr)-1)*sinθ2;
    for x := round(xl)-1 to round(xr)+1 do begin
        Shade Pixel(x,y) with
            PixelIntensity(yd1,yd2,pd1,pd2,θ1,θ2);
        pd1 := pd1 + pdinc1;
        pd2 := pd2 + pdinc2;
        end;
    end;
```

This algorithm does not make use of any simpler intersection cases between each pixel and the trapezoid. The x and y loop can be modified to do that by treating the first and last few cases in full generality and the middle elements which will not intersect the top and bottom edges as a special case.

7.2.2. Filling trapezoids using patches

The line drawing algorithms use small strokes to form lines; we can similarly use small patches to form trapezoids. By arguments used previously, we also need the ability to update a square area in order to achieve symmetric updates for all trapezoids. As during line drawing, there exist two strategies for using patches. We can either use parallel processing to compute individual patches or use precomputed patches to assemble the desired trapezoid. We shall discuss only computed patches; use of precomputed patches is fairly complicated although similar to the precomputed strokes technique of line drawing but is beyond the scope of this thesis.

7.2.2.1. Computing patches

As in the case of the incremental trapezoid filling algorithm, the universal polygon table will be used to compute the shades of individual pixels. The distance parameters will be computed using the method used to compute distances in the parallel computation algorithm used for drawing lines. Combining this parallel technique with an incremental algorithm which spans the whole trapezoid and incrementally computes the distances to the edges at the corners of each patch gives the following trapezoid filling algorithm.

```
θ1 := arctan((y1-y2)/(x1-x2));
θ2 := arctan((y1-y2)/(x3-x4));
yt := round(y1)-1;
yb := round(y2)+1;
xlinc := (x2-x1)*N/(y2-y1);
xrinc := (x4-x3)*N/(y2-y1);
xl := round(x1+xlinc)-1; xr := round(x2+xrinc)+1;

for yc := yt to yb by N do begin
    pdc1 := yc*cosθ1 - xl*sinθ1;
    pdc2 := yc*cosθ2 - xl*sinθ2;
    for xc := xl to xr by N do begin


    { The following nested loop is executed in parallel by the
        NxN processors }
        for x := 0 to (N-1) do
        for y := 0 to (N-1) do begin
             pd1 := y*cosθ1 - x*sinθ1 + pdc1;
             pd2 := y*cosθ2 - x*sinθ2 + pdc2;
             yd1 := y - y1;
             yd2 := y2 - y;
             Shade pixel (x,y) with
                PixelIntensity(yd1,yd2,pd1,pd2,θ1,θ2);
             end;

        pdc1 := pdc1 - N*sinθ1;
        pdc2 := pdc2 - N*sinθ2;
        end;
    xl := xl+xlinc; xr := xr+xrinc;
    end;
```

7.3. Conclusion

This chapter discussed algorithms to draw gray-scale lines and fill trapezoids. All the algorithms presented used geometric information to look up precomputed tables containing filtered pixel intensities. For both the tasks, an incremental algorithm is first discussed which updates the display one pixel at a time. These algorithms are modified to update several pixels in parallel. The parallel updates use two different methods: the pixels can either be computed in parallel or looked up from precomputed "fonts" of line strokes and trapezoidal patches. The precomputed font technique is easier to implement for high speeds but might not be feasible if the number of primitives is too large. Assuming the feasibility of a few hundred primitives, the precomputed font technique is implementable for both lines and trapezoids with the restriction that the corners have to coincide with pixel centers. If the corners do not coincide with pixel centers, then the parallel computation technique is more practical.

This chapter also restricts itself to using the conical filter with a radius of 1. In reality, the filter should be a parameter in the algorithms. The shape of the filter can be varied by varying the values in the filtering tables. Changing the extent of the filter requires a change in the algorithms, usually by changing the upper and lower limits of the loops.

The display design presented in Chapter 9 can be used to implement both algorithms although the precomputed font algorithms will be faster.

Chapter 8
Image Processing

Image processing refers to the manipulation of the pixels of an image to achieve desirable transformations. The image is represented by a two dimensional array of pixel values. Image processing algorithms process this array and perform a wide range of transformations on the image. Some simple transformations like translation, scaling, and rotation are used to present the image in a more desirable format. Another set of transformations is used to enhance the image to remove defects due to blurring or motion, or to improve features like contrast and brightness. Image processing algorithms are broadly classified into three classes: point-by-point transformations, neighbor transformations, and Fourier-domain transformations.

A point-by-point transformation is one in which the new image is computed by transforming each point of the image without referring to any other point. The simplest example of such a transformation is adding a constant to each pixel of the image to increase the brightness of the image. Similarly, multiplying each pixel by a constant increases or decreases the contrast of the image, depending upon whether the constant is greater than or less than one. Translation by an integer offset requires copying pixels from one location to another, and scaling by an integer factor requires copying pixels from one location to several. Rotations by multiples of 90 degrees can also be performed by merely moving pixels of the image. These point-by-point transformations and several others were the topic of discussion in Chapter 4 under the title of BitBlt.

Simple neighbor transformations are required to perform some image processing operations like non-integer translation and scaling, and non-perpendicular rotations. Such transformations typically combine the pixel intensities of a few neighbors to compute the pixel intensity of the new image. These operations together with their implementations are the topic of the first section of this chapter.

Fourier-domain transformations manipulate the image to achieve transformations in the frequency spectrum of the image. An example of such a transformation is edge enhancement, which corresponds to amplifying the high frequencies of the image. Other Fourier-domain transformations

are used to remove blurring and motion defects in digitization. Fourier-domain processing can be performed by convolving the image with the Fourier-inverse of the frequency domain transformation. Since most transformations manipulate the higher frequencies of the image, the convolution window is fairly small, usually a 3x3 or 5x5 or 7x7. Convolution is the subject of the second section of this chapter.

8.1. Neighbor transformations

This section discusses the algorithms of the neighbor transformations for non-integer translations and scaling, and for non-perpendicular rotation. The desire for such transformations is motivated by the same reason as the desire to draw lines between non-integer endpoints, which is to portray the gray-scale display as a device with a higher virtual resolution. This is especially important in moving images where the quantization to integers can cause additional aliasing in the form of jitter. An example of such aliasing could be a slow moving object on the display. If the object moves 2 pixels in 20 frames, and if the move is quantized to 1 pixel every 10th frame, then the object will seem to jump on the display. Having the ability to move the object by 1/10th of a pixel in each frame can be used to smooth the movement.

8.1.1. Translation

Translating an image on pixel boundaries is equivalent to shifting a sampled signal by a distance that is a multiple of the sampling rate, in which case the values of the individual samples remain the same except that they occur at a different spot. This operation can be performed by copying the samples from the original spot to the shifted spot (Figure 8-1). The *BitBlt* operator defined in Chapter 4 performed such an operation by copying a rectangular region from one part of the display to another.

Translation by sub-pixel distances is equivalent to shifting the sampled signal by a distance which is not a multiple of the sampling rate. This problem can be solved by reconstructing the original signal and sampling it again at the shifted intervals (Figure 8-1). To find an algorithm for this problem we should look at the sampling theorem and its implications.

Nyquist's sampling theorem states that a band-limited signal with a maximum frequency f can be sampled at a spatial interval of $1/(2*f)$ without losing any information contained in the signal which can hence be reproduced faithfully. Sampling signals that contain higher frequencies than allowed by

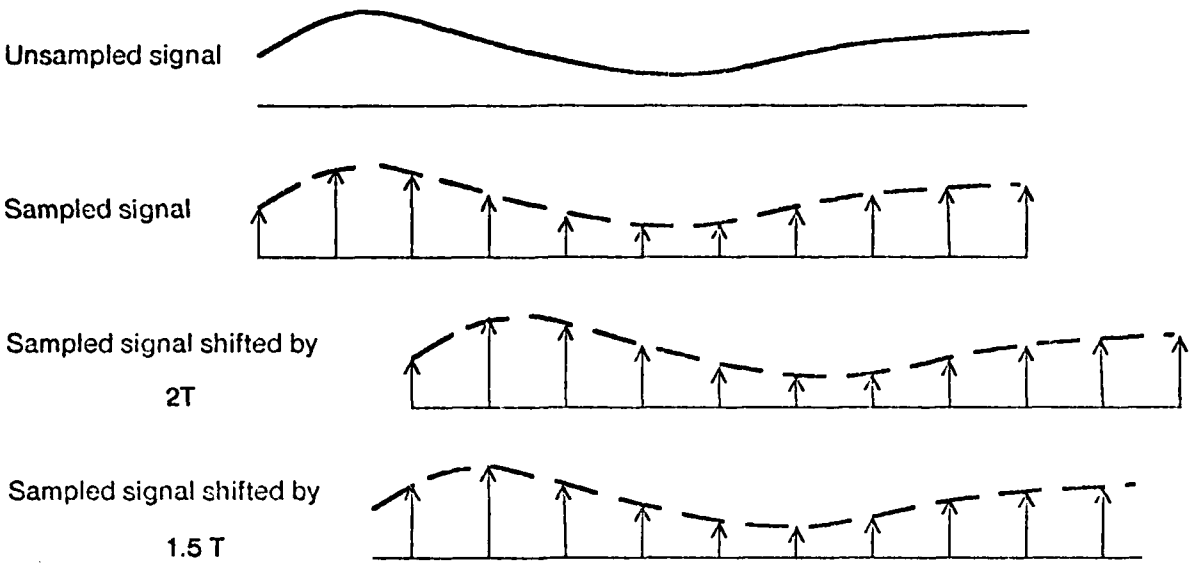


Figure 8-1: Sampling and shifting.

the Nyquist theorem result in the aliasing of the higher frequencies as lower ones. Figure 8-2 shows a high and low frequency both of which result in the same samples. Low pass filtering of the signal before sampling can remove the aliasing problem; this preprocessing step is hence called *anti-aliasing*. Ideally, a signal which is going to be sampled at time interval T should be prefiltered using a ideal low pass filter with an upper cutoff frequency of $1/(2*T)$. This corresponds to convolving the signal with the *sinc* function ($= \sin(\pi*x/T)/(\pi*x/T)$), which is the Fourier transform of the ideal low-pass filter. The resulting signal can then be sampled to create the anti-aliased samples.

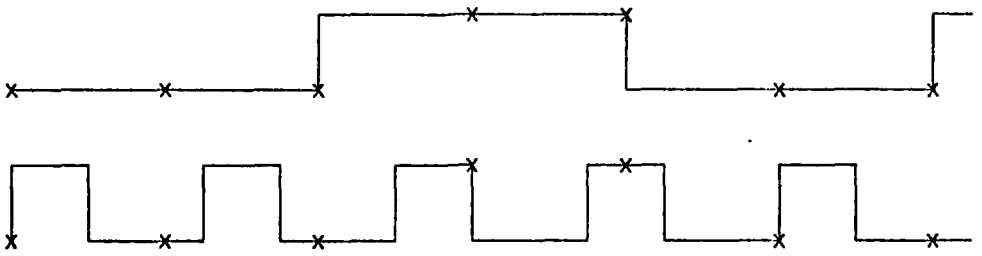


Figure 8-2: The aliasing problem.

The sample at time n can hence be computed as

$$S(n) = \int_{t=-\infty}^{t=\infty} S(t) \frac{\sin(\frac{\pi(t-n)}{T})}{\frac{\pi(t-n)}{T}}$$

$$= \int_{t=-\infty}^{t=\infty} S(t)\, f(t-n)$$

Shown in Figure 8-3 are the two signals that have to be convolved to compute the sample $S(n)$ at $t = n$. In practice, a computationally easier filter like the *triangular* or the *gaussian* filter is used for anti-aliasing (Figure 6-3), but these approximations introduce aliasing error by allowing some high frequency content of the original signal. Also some of the low frequency is lost because the frequency response of the filter is not perfect in the ideal low-pass range.

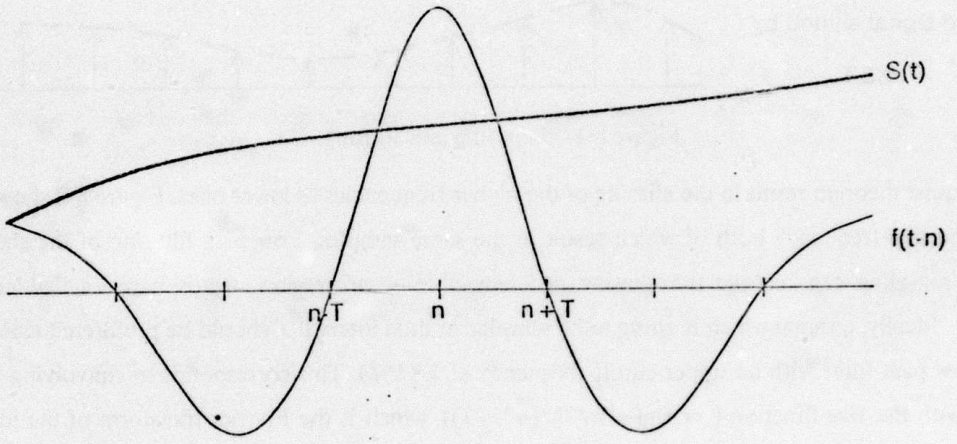


Figure 8-3: Filtering $S(t)$ to compute sample at n.

This set of samples have the same information content as the band-limited signal

$$g(t) = \sum_{n=-\infty}^{\infty} S(n) \frac{\sin(\frac{\pi(t-n)}{T})}{\frac{\pi(t-n)}{T}}$$

The *sinc* function is hence the ideal reconstruction function because there is no loss of information in the process. This yields us an algorithm that can be used to time shift the sampled signal by an interval which is not a multiple of the sampling rate. The sample at a time m is given by

$$S(m) = \sum_{n=-\infty}^{\infty} S(n) \frac{\sin(\frac{\pi(m-n)}{T})}{\frac{\pi(m-n)}{T}}$$

As in the case of filtering, this form of reconstruction requires the summation of an infinite sequence to create every sample. To make the problem computationally tractable, we can approximate the *sinc* function by one of several simpler functions. If we choose the *triangular* function then the reconstruction is simply a linear interpolation of adjacent samples. Shifting a sequence by half the sampling interval can hence be performed by averaging adjacent samples. The shift of an interval of α/T can be done by computing

$$S(n') = S(n)(1-\alpha) + S(n+1)\alpha.$$

This linear interpolation can be extended to two dimensions by interpolating four adjacent neighbors.

The linear interpolation algorithm for reconstructing signals from their samples causes a loss of information, because in the ideal low-pass band, the triangular filter does not have a linear frequency response (Figure 6-3). Successive interpolations will hence blur the image due to further loss of high frequency information. However the first few interpolations give satisfactory results.

Figure 8-4 shows the effect of successive interpolations on a sampled character. The lower left corner contains adjacent characters "*A*" created by filtering a high resolution representation of the character [Warnock 80]. In the pair just to the right of the pair in the lower left corner, the first character is an identical copy of the original, and the second character is moved to the left by 1/16th of a pixel by interpolating the original. The pairs further to right have one original and the second character moved horizontally 2/16 pixel, 3/16, ... and so on until the second character of the last pair is moved by one full pixel. All moves are made by successive interpolations, i.e. the character with an offset of 3/16 is created by moving the character with an offset of 2/16 by a distance of 1/16. The vertical column on the left is similar to the bottom row with the difference that the second character is moved vertically instead of horizontally. The other rows and columns are moved both vertically and horizontally with horizontal displacement determined by the column number and the vertical displacement determined by the row number. Each pair of characters is created by interpolating the pair on its left, and if there is no set on the left then the set below is used. The figure shows that the characters become unacceptable after approximately five or six successive interpolations.

Figure 8-5 shows the same character placements as Figure 8-4 with the difference that now all characters with sub-pixel positioning are created from the master copy located at the lower left corner. The result of this experiment is that interpolation is a satisfactory way of translating images by

sub-pixel distances if only the master copy is used. Successive interpolation will blur the image and render it unsatisfactory.

One practical implication of the interpolation technique is that gray-scale fonts need only to store one character positioned at a pixel center; interpolation can be used to get sub-pixel positioning. Interpolation hence reduces font space used by text and graphics fonts. (The graphics fonts are used to store precomputed strokes for lines and patches for trapezoids.)

The interpolation algorithm relies on the linear reconstruction filter. Higher order filters will result in smaller distortion; the *sinc* filter gives the ideal reconstruction. Such filters rely on computing a function of more than two neighbors and are hence more expensive, but also result in negative values for some samples which are very hard to interpret and show on displays!

Parallel implementations of the interpolation algorithm require access to both rows and columns in order to achieve a symmetrical speedup. The square memory organization provides such access. The procedure used is identical to the BitBlt procedure with the *operation* replaced by the interpolation of adjacent pixels. The neighbor connections are used to access the values of neighboring pixels. Since any of eight neighbors might be required, the tri-state neighbor connections (Figure 4-14) are most suitable for this application.

8.1.2. Scaling

Scaling is a commonly used operation to view an image at different sizes. Integer scaling can be performed easily by replicating pixels. Such an operation is often not sufficient, especially in motion sequences where gradual scaling is required. Non-integer scaling redistributes the intensity of individual pixels. The scaling problem can be translated into an equivalent signal processing problem of reconstructing the original from its samples, scaling the original signal, and then resampling the scaled signal. Figure 8-6 shows the scaling of a signal by a scale of less than and greater than one. The figure uses the triangular reconstruction for simplicity instead of the ideal *sinc* reconstruction function. This leads to an obvious one-dimensional algorithm for scaling an image with n pixels by a factor of α.

```
for x := 0 to ⌈α*n⌉ do begin
    xinv := x/α;
    xi := ⌊x/α⌋; xf := x/α - xi;
    I'[x] := I[xi]*(1-xf) + I[xi+1]*xf;
    end;
```

This algorithm uses real representations of α, *xinv*, and *xf*. An integer algorithm can be formulated if

Figure 8-4: Character dragged by successive moves.

Figure 8-5: Character moved from master copy.

we assume α to be representable as fraction p/q of small integers. The scaling problem can now be restated as the redistribution of p parts of each original sample into q parts of each scaled sample (Figure 8-7) [Lucas 81]. This algorithm can be stated as

```
for x := 0 ⌈p*n/q⌉ do
    I'[x] := 0;

for i := 0 to p*n do
    I'[i/q] := I'[i/q] + I[i/p]/p;
```

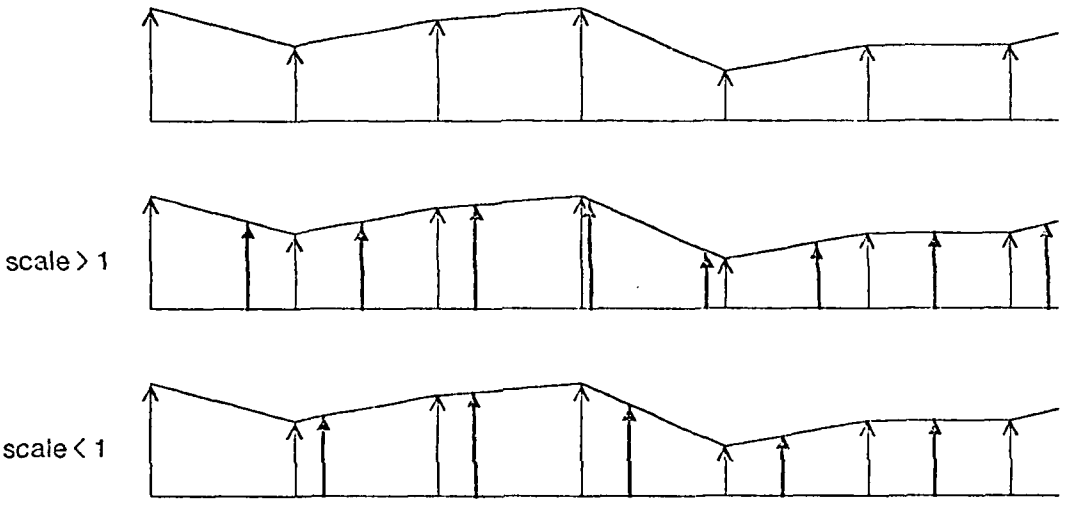


Figure 8-6: Scaling an one-dimesional signal.

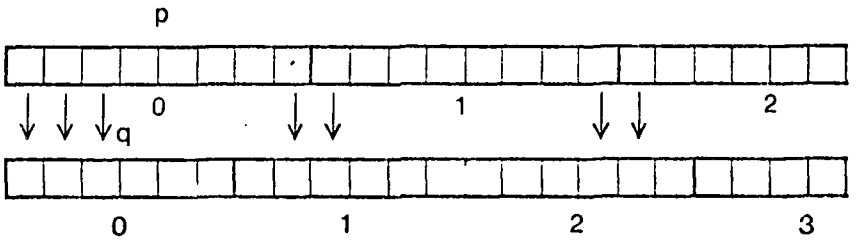


Figure 8-7: Scaling using integers.

Parallel implementations of the scaling algorithm are very hard because there does not exist a one-to-one mapping from the pixels in the source image to pixels in the destination image. However, if we are scaling a two-dimensional image in only one dimension, then we can operate on pixels in the other dimension in parallel. The NxN memory organization will provide a N-fold speedup for images scaled in only one of two dimensions. Scaling transformation for both dimensions can make two passes over the image, one to scale vertically and one to scale horizontally.

8.1.3. Rotation

The BitBlt discussion in Chapter 4 discussed rotations by multiples of 90 degrees and memory architectures to implement them. These operations are similar to integer translation and scaling because they can be performed by copying pixel intensities. Rotations by other angles not only cause the pixels to get mapped in a non-linear way, but also require the intensity values to be redistributed. Figure 8-8 shows the mapping of destination pixels to the source region when an image is rotated by 45 degrees. The dots show the locations of the original pixel array, and the circles show the locations of the rotated pixel array. Linear interpolation can be used to compute the intensities of pixels that do not lie on the source pixel grid.

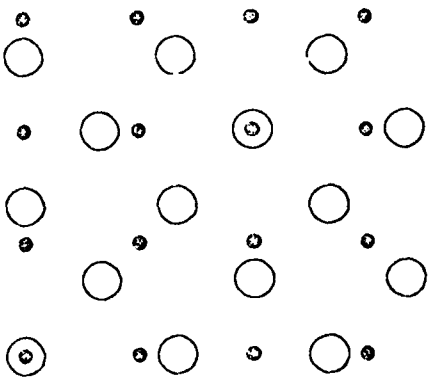


Figure 8-8: Rotation by 45 degrees.

Rotation can be performed by using a combination of shearing and scaling transformations. A shearing transformation maps a rectangular image into a parallelogram, leaving the orientation of one of the parallel edges unchanged (Figure 8-9). These transformations are described by characteristic matrices which 2x2 matrices used to multiply the old (x,y) coordinates to obtain the new coordinates. Scaling transformations have characteristic matrices of the form

$$\begin{bmatrix} sx & 0 \\ 0 & 1 \end{bmatrix} \ (\textit{horizontal}) \ \text{and} \ \begin{bmatrix} 1 & 0 \\ 0 & sy \end{bmatrix} \ (\textit{vertical}),$$

and shearing transformations have characteristic matrices of the form

$$\begin{bmatrix} 1 & v \\ 0 & 1 \end{bmatrix} \ (\textit{vertical}) \ \text{and} \ \begin{bmatrix} 1 & 0 \\ h & 1 \end{bmatrix} \ (\textit{horizontal}).$$

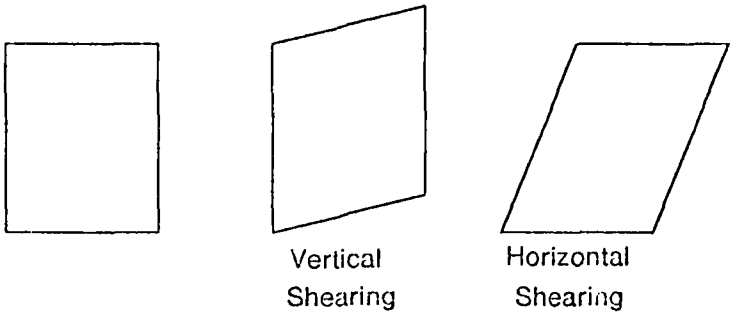


Figure 8-9: Shearing.

Shearing and scaling can be used in combination to rotate an image with an arbitrary angle r, because

$$\begin{bmatrix} \cos r & \sin r \\ -\sin r & \cos r \end{bmatrix} = \begin{bmatrix} 1 & \tan r \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -\sin r \cos r & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \cos r \end{bmatrix} \begin{bmatrix} \frac{1}{\cos r} & 0 \\ 0 & 1 \end{bmatrix}.$$

Carl F.R. Weiman [Weiman 80] presents an algorithm to implement the shearing transformation by using a code devised and studied by Rothstein [Rothstein 79] for a transformation representable as p/q. Rothstein's code is a binary sequence, the digits of which correspond to jumps in a bit-map line with slope q/p. The code will have a 1 when this line jumps and a 0 whenever it does not. As an example, the code for 4/3 will be (0111) when the origin of the line corresponds to a pixel center. For lines whose origin do not correspond to pixel centers, similar codes can be constructed. Other possible codes for 4/3 are (1110), (1101), and (1011). The algorithm performs the shearing by shifting up each column (for vertical shearing) where the code is 1, doing this for every possible code, and averaging the results (Figure 8-10 which is taken from [Weiman 80]).

Weiman's algorithm has a BitBlt implementation in which each column (again for vertical shearing) gets added to its destination followed by dividing the intensity of each in the result by q.

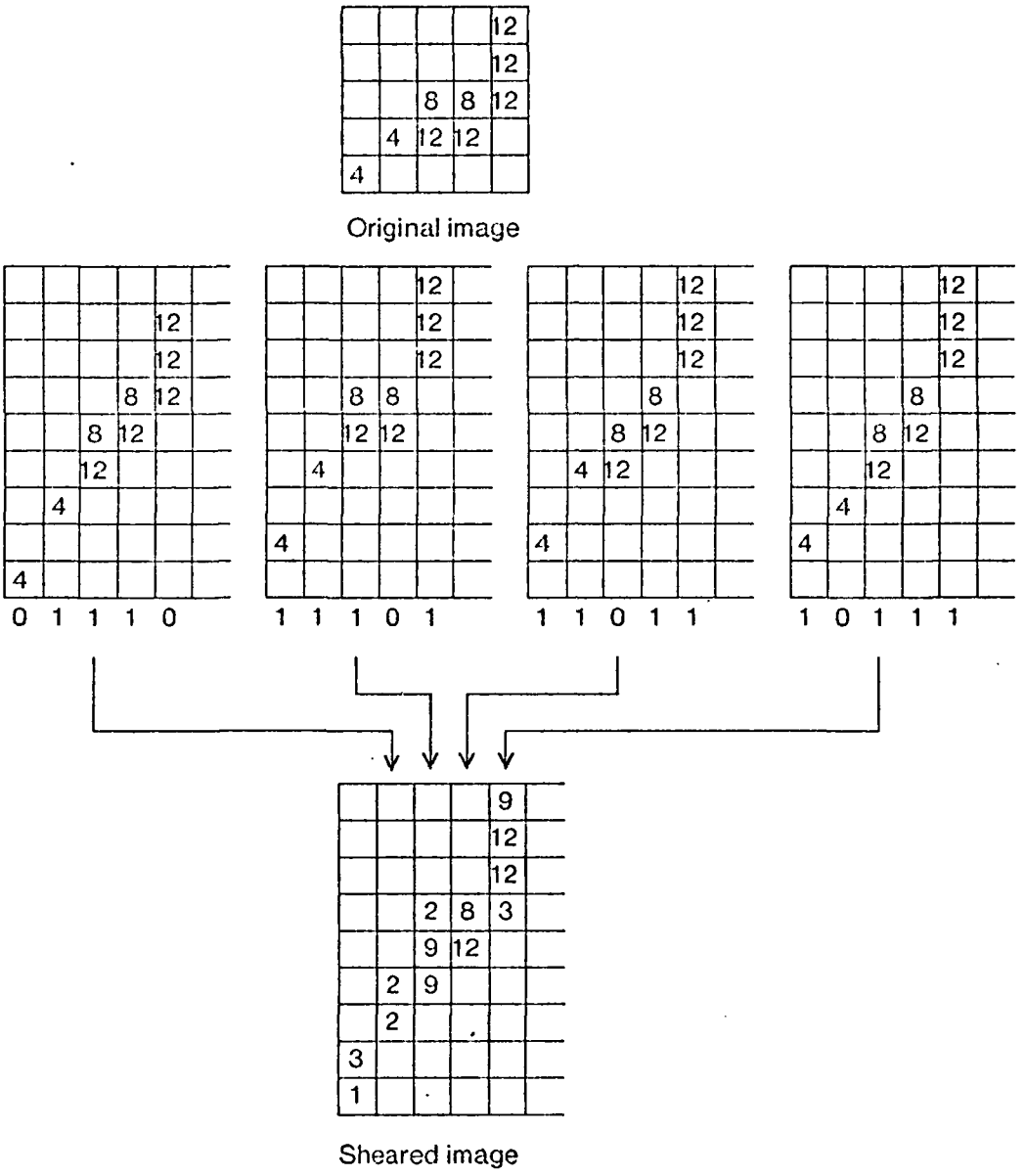


Figure 8-10: Vertical shearing of 4/3 using Rothstein's code.

8.2. Convolution

A transformation which changes the high-frequency characteristics of an image can be implemented as a convolution of the image with a set of coefficients which are zero everywhere except in a small window. Such convolutions can be used for operations like smoothing, edge detection and enhancement, spatial reduction, or any other operation which changes only the high frequency characteristics of the image. We shall discuss edge detection as an example.

Edge detection is used during image processing for picture segmentation. It segments the picture into regions based on the detection of discontinuities in the intensity level. Such a discontinuity is called an edge and there are several techniques used to detect such edges. Most of them use derivative operators, which compute the gradiant of the image intensity, and hence give high values at points where the intensity level of the picture is changing rapidly. One of the popular digital gradient approximations is due to Roberts [Roberts 65], which computes the following function of each pixel (x,y)

$$\max(|I(x,y)-I(x+1,y+1)|,|I(x+1,y)-I(x,y+1)|).$$

This function relies on the fact that differences in any pair of perpendicular directions can be used to compute the gradient.

A parallel implementation of Roberts' function or any 3x3 convolution requires each pixel to access its nearest eight neighbors. It can read these values and combine them with its own to compute the function. This parallel approach will work even with NxN squares (if $N \geq 3$), the only problem being that pixels on the edges cannot get access to all their neighbors. This can be solved by overlapping successive squares by one pixel in both directions.

An outline of the algorithm to perform Roberts' function follows. Place the initial NxN square at the top left corner of the image. All pixels except the ones on the lower and left edges will compute the convolution function. Move the NxN square right by (N-1) pixels and recompute. Repeat until the right edge has been reached. Now place the NxN square at the left edge (N-1) pixels below the previous left edge placement. Repeat these two nested loops until the whole image has been processed. Notice that this algorithm is an optimization of the most obvious algorithm which overlaps the squares by two pixels and performs no computation for all edge pixels. This algorithm can be easily extended for larger windows.

Convolution, like other image processing applications relies on neighbor interconnections for speed.

8.3. Conclusion

This chapter presented algorithms for a few low level image processing applications that can be implemented effectively for the NxN square memory organization. The intent of this presentation is *not to present the best algorithms for these applications*, but to show that the square memory organization is indeed effective for image processing in addition to being well suited for BitBlt and graphics.

Chapter 9
Display Design

The architectures and algorithms developed in this thesis lead to one conclusion: the critical component of a display design is the underlying memory system. The memory system design determines the upper limit of the speed at which the display can be updated. This thesis discusses several memory organizations; the symmetric square organization proves to be most effective to provide parallel updates for BitBlt, graphics, and the image processing applications.

The MxM square memory organization is defined to use M^2 memory chips which can be read or written simultaneously. But the term *memory chip* has always been used loosely. In Chapter 2, each memory chip could compute addresses from the location of the display region being accessed. It could also compute masks to disable a subset of the memory chips. In Chapter 4, the memory chips were able to communicate data to other memory chips. They could also perform simple arithmetic and logic operations required by BitBlt. In Chapter 7, each of the memory chips contained a processor which could perform distance computations and use these distances to lookup anti-aliasing tables to determine pixel intensities. Finally, in Chapter 8, we abandoned the hope of each memory chip having only a small special purpose processor by providing fairly general operations.

None of these *memory chip* requirements is met by any commercial memory part. One possible structure of a *smart display memory chip* that meets these requirements is shown in Figure 9-1. Since each memory chip is defined to have the ability to read or write one pixel in every memory cycle, the heart of the smart display memory chip is a pixel RAM. The pixel RAM is a conventional random access memory which provides as many bits in each memory cycle as the number of bits in each pixel. The processor associated with the pixel RAM has exclusive control of the address and write enable wires of the RAM and exclusive access to the data. The interprocessor communication wires provide the data communication required by BitBlt and the image processing applications. The video buffer aids the generation of video and is discussed in Section 9.1.1.

Present technological limitations do not allow the fabrication of a *smart display memory chip* as a

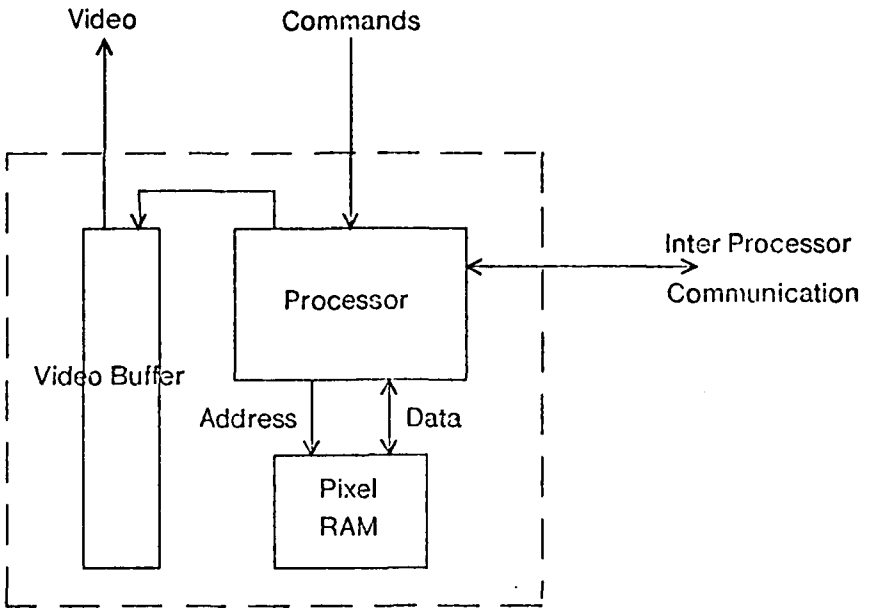


Figure 9-1: Smart display memory chip.

single chip. However the RAM can be separated from the rest of the chip as shown in Figure 9-2. This allows the use of commercial RAM parts and still requires only one chip to be custom designed. The processor chip used in such a scenario is referred to as the *display chip*.

This chapter discusses the design of a display memory system for the 8x8 memory organization. Sections 9.1, 9.2, and 9.3 explore the design space. Section 9.4 describes *DChip*, a display chip designed for such a display system. The *DChip* design is optimized for BitBlt, although it is also able to perform the computations for the other operations discussed in this thesis. Section 9.5 presents simulated instruction sequences of how *DChip* can perform various operations. These simulations are then used to predict the performance for these operations. Section 9.6 discusses issues about scaling the design of *DChip* with respect to various parameters.

9.1. Memory organization

The 8x8 memory organization has several implementation choices. The 8x8 squares can be staggered (see Section 2.3) to allow the access of both 8x8 squares for the update operations and 64x1 spans for the video scanning. This choice is discussed in Section 9.1.1. To provide effective bandwidth utilization for both BitBlt and precomputed graphics algorithms, we choose to provide the ability to update arbitrarily aligned 8x8 squares. We shall, however, restrict ourselves to updating

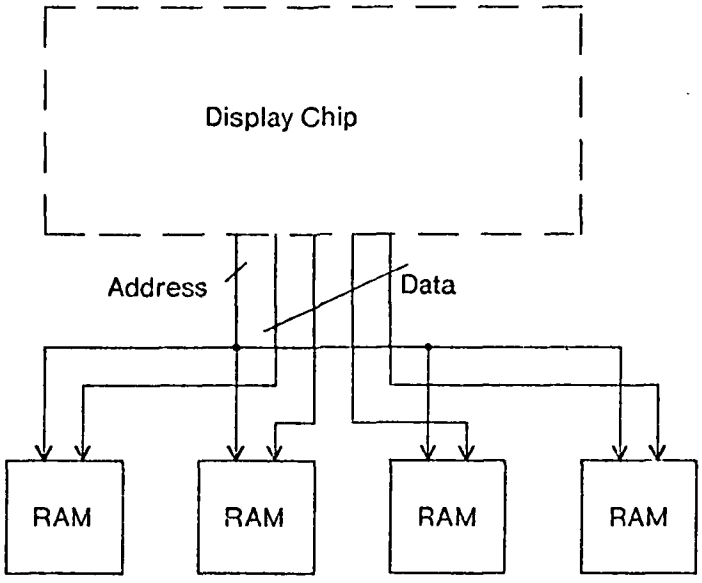


Figure 9-2: Display chip with 4 RAM chips for a 4 bit gray-scale system.

contiguous display areas. As a result different parts of the memory array will receive different addresses. The various choices for computing the memory addresses are discussed in Section 9.1.2.

Jim Clark and Marc Hannah use an addressing scheme in which each memory chip receives an independently computed address, as a result of which the display area accessed is not necessarily physically contiguous [Clark 80]. Their scheme results in a design which is substantially more complicated than the one discussed in this chapter.

9.1.1. Screen refresh

In one memory access the unstaggered 8x8 organization provides only 8 pixels of the scan line being scanned. Because of the desire to waste the smallest fraction of bandwidth for the video scanning process, we have to save the remaining 56 pixels in a separate piece of memory. This memory is called the *video buffer*, and contains storage for at least 8 complete scan lines. Having storage for only 8 scan lines forces the video controller to carefully time the writing and reading of the video buffer, which may be very difficult if the access time of the video buffer is only slightly higher than 8 times the video rate. However, if the video buffer has storage for 16 scan lines, then the video controller can write one set of 8 scan lines while reading the other (i.e., double buffering). Also, the frame buffer can be read in a burst which accesses all the 8 scan lines at once or in smaller bursts.

Burst access of the unstaggered 8x8 organization along the scan line direction requires only the column address to be incremented by one (Section 2.2). This allows the use of *page mode* provided by commercial RAMs, which saves a factor of two in the frame buffer bandwidth used for video scanning. To avoid concurrent access of the video buffer for reading and writing, 1/8 of the next 8 scan lines can be written during each horizontal retrace because no reads are required during that period. In such an implementation only 1/8 of the total number of 8x8 square along a scan line are read during each frame buffer burst access.

Only 8 pixels out of the 64 available during the access of the video buffer are used to generate the next 8 pixels of video. This implies the use of multiplexing to select the appropriate 8-pixel row. The multiplexing can be implemented by using 8 external 8x1 multiplexers[7]. The presence of output enables on the outputs of the video provides the same multiplexing without the use of external multiplexers. The video generation from the video buffer can be implemented as shown in Figure 9-3.[8]

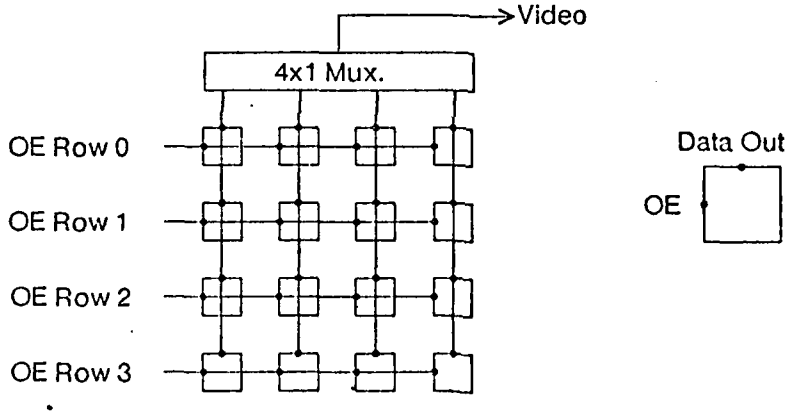


Figure 9-3: Video generation.

The 8x8 display implemented the video buffer using RAMs, although it does not need the generality of random access. Both the reading and writing are serial in the same direction, which suggests the use of shift register memory. Figure 9-4 shows a double rail shift register, 64 copies of which can be used as the video buffer. It consists of two dynamic shift registers which shift in

---

[7] Assuming single bit video. If the video has several bits of gray-scale, then these numbers are multiplied by the number of bits in the gray-scale.

[8] The figure only shows a 4x4 square, but is easily extended to an 8x8.

opposite directions. The length of each is 1/8 of the length of the scan line. The data in the Input Shift Register can be transferred in parallel to the Output Shift Register. Only the input to the Input Shift Register and the output of the Output Shift Register are accessible. This buffer is used by shifting in data serially into the Input Shift Register, transferring all the data to the Output Shift Register, and then shifting it out in the reverse order. To use it as a video buffer the scan line would be shifted in in reverse order, transferred to the Output Shift Register, and read out as video. The reason for the shift registers to run in opposite directions is to use the buffer for displays with different scan-line lengths. If the size of the scan line is smaller than the maximum, we can just ignore the unused part of the shift register. The video buffer can be fabricated as part of the display chip and requires a total of $g+1$ extra pins, where g is the number of bits in the gray scale. Even though a shift register may be used instead of RAM as the video buffer, a fairly large area has to be devoted to its implementation. For a g-bit gray scale system with 768 pixel scan lines, the video buffer stores $12g$ kilobits, or $192g$ bits in every display chip.

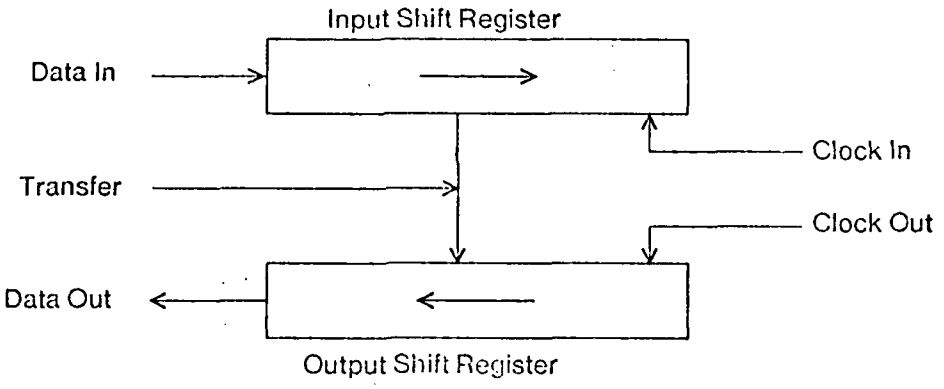


Figure 9-4: Video buffer.

The staggered addressing allows the access of 64 horizontal pixels, which eliminates the need for the video buffer (see Section 2.3). The multiplexing shown in Figure 9-3 can still be used to generate the video. Burst accesses of the frame buffer cannot take advantage of the page mode memory access. Burst access may still be useful for the overall system design because it implies a lower frequency of swapping between the update and video refresh tasks. For example, each display chip could have a 12-pixel FIFO buffer which could be filled during a burst access of the frame buffer and used for refreshing a 768 pixel scan line.

To effectively utilize the memory bandwidth of any organization, we have to implement fast address computations. *Ease of implementation for the address computations caused the choice of the*

unstaggered square organization. This organization forces the use of a video buffer, but also offers higher update bandwidth because of the ability to use page mode accesses for screen refresh. The discussion in the rest of this chapter refers to the unstaggered 8x8 organization.

9.1.2. Memory addressing

Allowing for the accesss of arbitrarily aligned 8x8 squares implies the addressing of different parts of the memory chip array with different addresses. If the display is built using *smart display memory chips*, then the address computation can be performed on-chip and the external world would provide only the location of the square being accessed. We are, however, going to use commercial memory with *display chips*, and have two alternatives for generating addresses for the memory. We can either use the display chips to generate the addresses or alternatively generate the addresses outside the display chips and directly drive the address wires of the memory chips. The first alternative is identical to the smart display memory approach.

If the addresses for the memory chips are generated externally, and the display chips are used only for data, then there are several possible alternatives to implement addressing. The most obvious implementation of the addressing requires 64 sets of address wires and a large amount of multiplexing hardware to select the appropriate address on these wires. However, a simple space-time tradeoff can implement this addressing more economically by using eight sets of address wires bussed down the columns of the memory chip array (Figure 9-5). If each memory chip is a 64K RAM, then each address bus has eight address wires to carry either the row or the column addresses. The RAS (Row Address Strobe) lines are bussed across the rows and the CAS (Column Address Strobe) lines are bussed down columns. Three addressing steps are required to provide each chip with the correct address. To address the square located at $(x*8+i,y*8+j)$, the first step places y on all the address busses and strobes the RAS lines on $(8-j)$ rows from the bottom. The second step places $y+1$ on all the address busses and strobes the RAS lines on j rows from the top. Now all the chips have the correct row address. The third step places $x+1$ on the first $(8-i)$ address busses and x on the rest of the i address busses and strobes the CAS lines on all the chips. This implementation uses only 8 address busses instead of 64 but has to use three cycles instead of the usual two to deliver a full address. This mechanism was used in the 8x8 display.

Alternatively, the location of the square to be accessed can be broadcast to all the display chips which can then individually compute the row and column addresses using the following algorithm discussed Section 2.2. Assuming (x,y) is the location of the top left pixel of the 8x8 square, then:

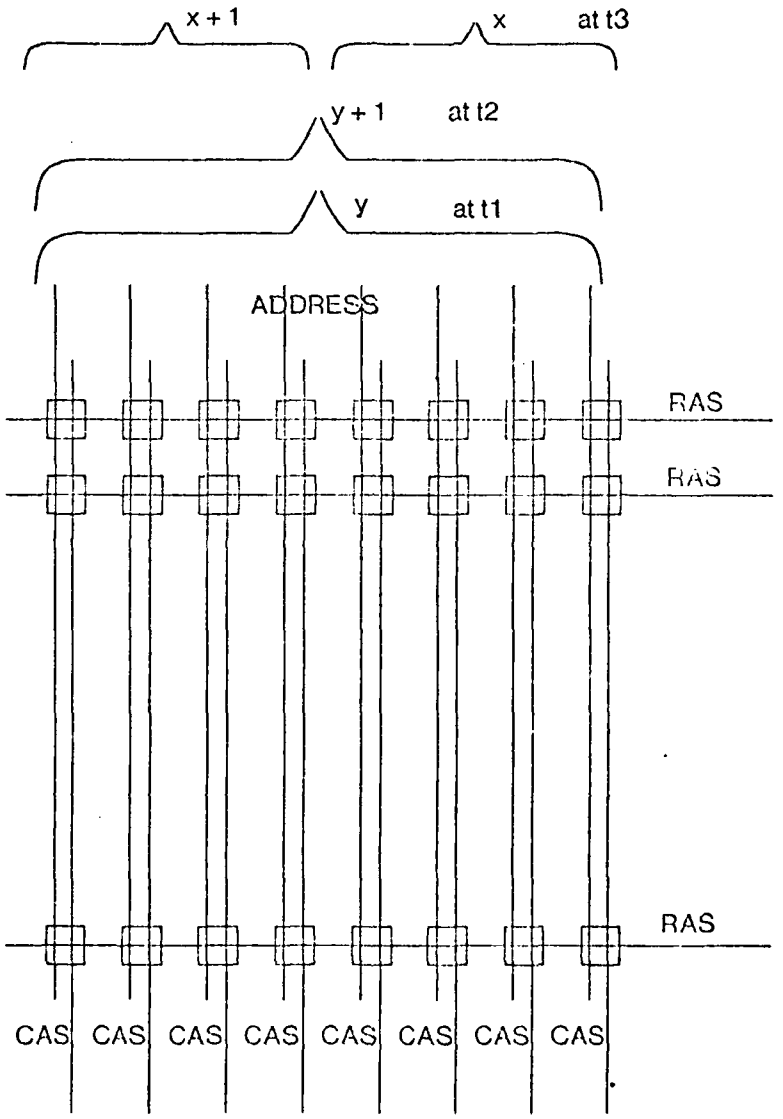


Figure 9-5: Addressing the memory without the use of display chips.

$CA(x,y) := x/8;$
$RA(x,y) := y/8;$
$i := x\%8;\ j := y\%8;$
if $(c < i)$ then $CA := CA + 1;$
if $(r < j)$ then $RA := RA + 1;$

where c and r are the column and row positions of the display chip. They are prestored constants in each display chip. These addresses are easily updated for subsequent squares in the close proximity of the original one. The address for the square at $(x+8,y)$ can be updated by

$$CA(x+8,y) := CA(x,y) + 1,$$

and the address for $(x,y+8)$ can updated by

$$RA(x,y+8) := RA(x,y) + 1.$$

The discussion of BitBlt in Chapter 4 showed that large area BitBlts udpate squares in close proximity to each other. The easy address updates for such squares imply that the initial address computing procedure can be slow and may indeed be performed as a sequence of instructions. But small area BitBlts (e.g. characters) and the graphics algorithms that use precomputed strokes and patches still require fast address computations.

The display chip can use a hardwired circuit to compute the row and column addresses. The computing of row and column addresses use identical algorithms, and because these addresses are used at different times, one circuit will suffice to compute both.

An advantage of storing x and y in all display chips is the increase in performance when only one of the parameters changes although this is more expensive because each display chip devotes space for storing duplicate information. A different possibility is to store the x and y externally and broadcast them to the display chips only when the addresses are computed. Figure 9-6 shows this technique in the form of a block diagram. Although the algorithm is insensitive to whether the row or the column address is being computed, a single bit of information is required to determine whether to use the row or the column number in the comparison. A simplification of this scheme performs the comparison externally and instructs each display chip whether or not to increment the address (Figure 9-7). During the column address phase, all display chips in the same column receive the same value of *CInc*. Similarly during the row address phase, all display chips in the same row receive the same value of *RInc*. The *CInc* control pins can hence be bussed vertically and the *RInc* control pins can be bussed horizontally (Figure 9-8). During the column address phase, the ROM on the top of the chip array will use the value of $x \% 8$ to compute the values of the *RInc* wires that are enabled vertically to the display chips. Table 9-1 tabulates the contents of this ROM. The same operation is executed by the ROM on the left edge during the row address phase.

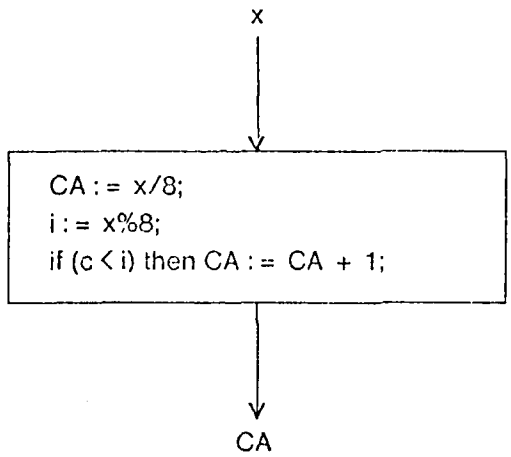


Figure 9-6: Address computation by each display chip.

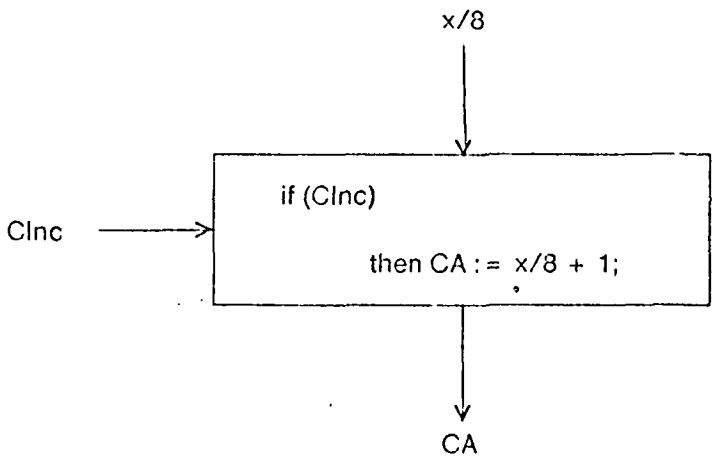


Figure 9-7: Simplified address computation.

9.1.3. Masking

Each display chip controls the write-enable signals of its memory chips. This allows selective update in two different ways. First, any subset of the display chips may enable their memory chips to update only part of an 8x8 square. Second, each display chip may enable a subset of its memory chips to selectively update only a subset of the bit planes. In order to reduce pin count of the display chip, the second capability has been eliminated from the present design. However, selective update of bit planes is still possible by using a read-modify-write cycle.

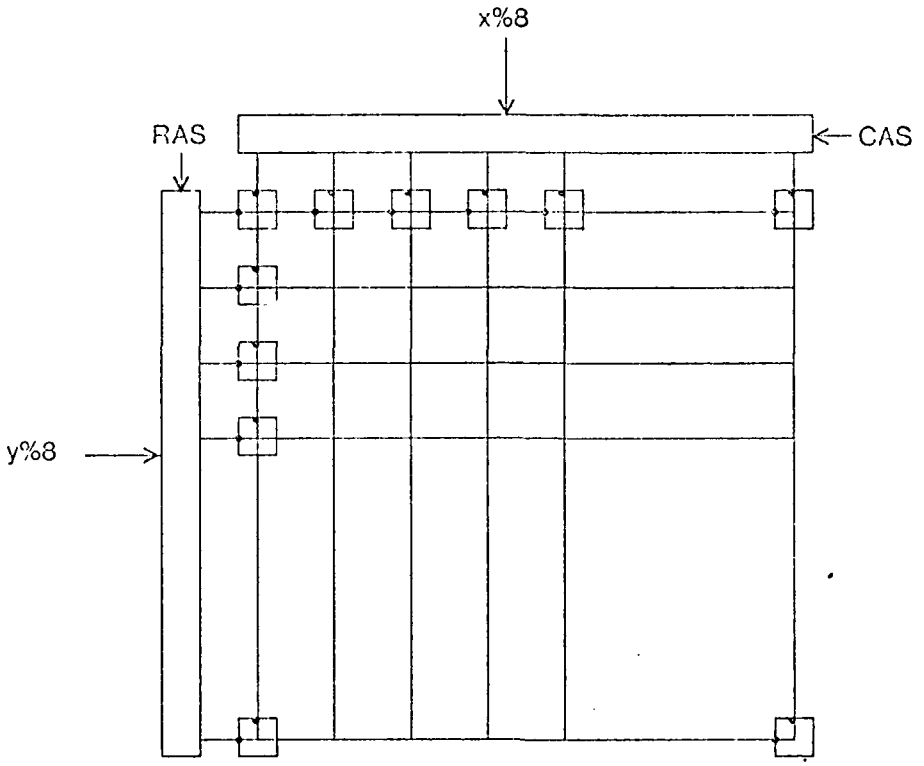


Figure 9-8: Controlling the INC pins for address computation.

x%8 or y%8	*Inc* Vector
0	00000000
1	10000000
2	11000000
3	11100000
4	11110000
5	11111000
6	11111100
7	11111110

Table 9-1: *Inc* vector used for different values of the offset from the 8x8 boundary.

The write enable masks that control updates to parts of the 8x8 square are often rectangular. Such masks arise during the end condition of BitBlt rectangles or lines whose lengths and widths are often not multiples of 8. The mask may not be rectangular when a non-rectangular pattern is being inserted. In such cases the raster data of the pattern is used as the mask.

The rectangular masks can be forced to be aligned with at least one of the corners of the 8x8 square because the memory organization allows the access of arbitrarily aligned squares. As shown in Figure 9-9, the 8x8 mask can be formed by ORing two 8-bit masks (called *xmask* and *ymask*), one for each dimension. The 8 bit *xmask* contains zeroes in the columns that have to be enabled and ones in the columns to be disabled[9]. Because the masks are used only for corner aligned rectangles, the zeroes are contiguous and flushed either to the left or the right. There are 16 such masks for each dimension, a total of 32 masks. These masks have to be aligned with the display memory's word boundaries. For example, if the update is not aligned with 8x8 boundary, then the top left corner bit of the mask has to be used in the chip that contains the top left corner of the square being updated.

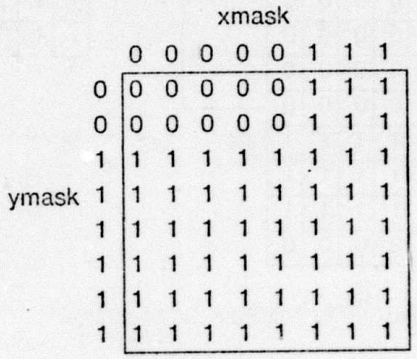


Figure 9-9: Masking 8x8 squares using *xmask* and *ymask*.

As in the case of addresses, these masks can be computed. Four parameters are needed to compute each mask: the count of zeroes in each direction and the boundary offsets of the square being updated. The mask computing algorithm will not be discussed; instead a table lookup technique is used to access the precomputed masks stored in the frame buffer. Two memory accesses are required to access each mask (Figure 9-10). The first memory access uses three parameters: *xmask* is number of columns enabled, *x*%8 is the x-offset of the square being updated from the 8x8 square boundary, and *xdir* is 0 if the enabled columns are left aligned and 1 if they are right aligned. The second memory access uses the y-directional parameters. The two masks are then ORed to produce the final mask.

---

[9] A zero as the write enable bit enables the write operation.

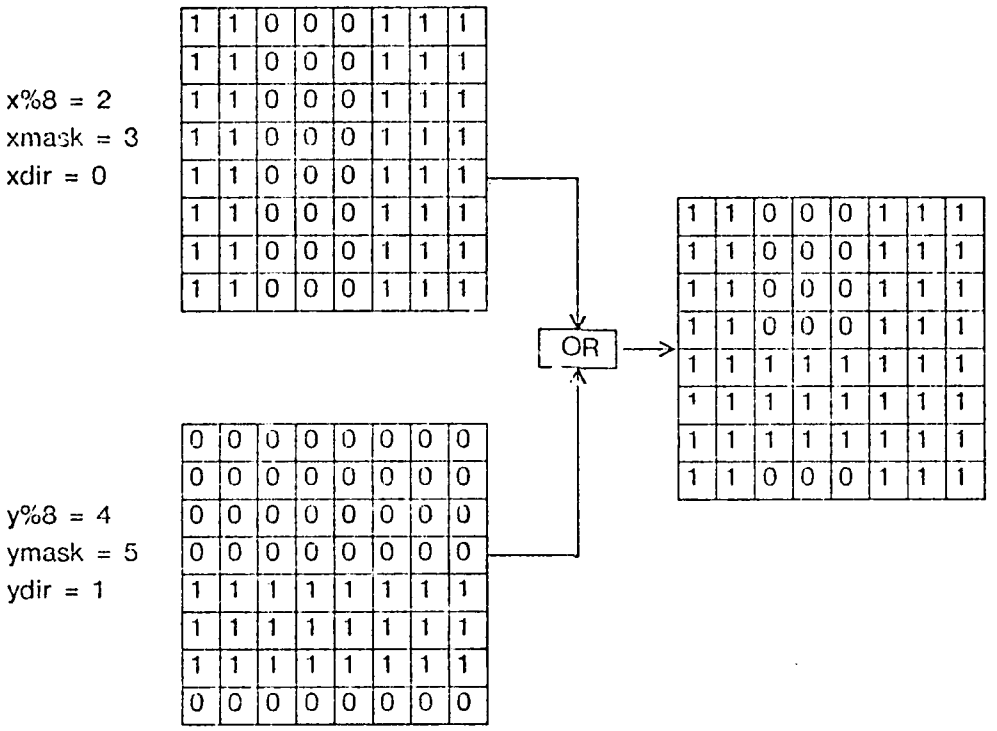


Figure 9-10: Masking 8x8 squares.

9.2. Interprocessor communication

The array of display chips have to communicate with each other for two different reasons: data alignment for BitBlt applications and neighbor access for the image processing applications. The communication pattern for BitBlt is simply a two-dimensional circular shifting of the data. It can use one of several mechanisms proposed in Chapter 4. The faster mechanisms require more pins than the slower ones.

Figure 9-11 shows a display chip with four tri-state paths such that, during one cycle, the display chip can enable its data on any one of the paths and can also latch the data from any of the remaining paths. This requires $4g$ pins for data (g is the number of bits in a pixel), and four pins for control: two to specify the input direction and two to specify the output. This configuration can be used to implement all the interconnection mechanisms discussed in Chapter 4. In the case of mechanisms that use only two paths like the neighbor connections in the one-dimensional organization and the sequential connection in the square organization, only two paths are bonded to implement these

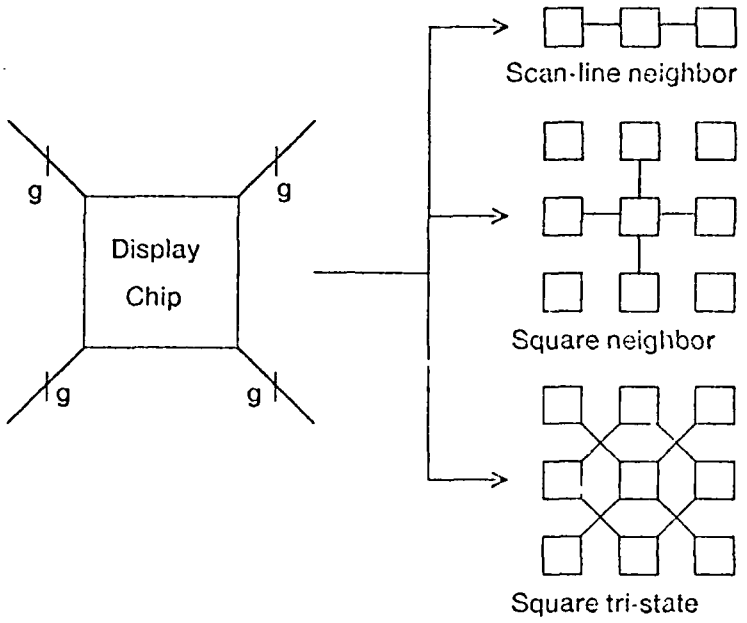


Figure 9-11: Four tri-state paths for each display chip.

mechanisms without the use of extra pins. The proposed configuration is ideally suited for the tri-state communication, which requires three control pins to specify the direction of transfer. This configuration hence uses one extra pin for control, but provides the versatility of being usable for other interconnection mechanisms.

The tri-state paths allow access to any of the eight neighbors in each cycle. The image processing algorithms discussed in the previous chapter use neighboring pixel intensities and can be implemented efficiently using the four tri-state data paths.

9.3. Processor

The computations required of the display chip are as follows:

1. BitBlt uses elementary boolean operations for bit-map images (Table 4-1), and simple arithmetic operations for gray-scale images (Table 4-2).

2. The parallel algorithm to compute line strokes computes the perpendicular distance to the line (Figure 7-4). This distance computation requires the multiplication of numbers in the range between 0 and 7. This multiplication can be implemented using three sequential additions, although a single-step implementation uses only a 3-input adder. The same algorithm is used for computing trapezoidal patches. The distance computation

is followed by a table lookup to find the anti-aliased intensity of individual pixels. This table can be part of the processor but we choose to store it in the frame buffer memory. This choice has two implications. Firstly, the frame buffer memory must be larger than the display area. Secondly, the processor must be able to modify the addresses of the frame buffer memory with a computed value, which in this case is the perpendicular distance.

3. The algorithms for sub-pixel translation and non-integer scaling require linear interpolations. Linear interpolation is a computation of the form $(A\alpha + B(1-\alpha))$, where A and B are two intensity values and α is real number less than one. If α can be represented as a fraction p/q such that q is power of 2, then the interpolation can be implemented as a series of additions followed by right shifts.

For the applications discussed in this thesis, the display chip processor can perform these computations with an extremely simple ALU together with a few registers to store temporary values.

9.4. Display chip

This section describes a *display chip* (DChip) that is designed for a 4-bit gray-scale 8x8 display system. An 8x8 display system will use 64 copies of this display chip. Each display chip interfaces to four 64K RAM chips. The chip provides tri-state connections to four adjacent neighbors. Due to the lack of pins, the chip provides only a video buffer for single-bit video. A block diagram of this chip is shown in Figure 9-12, and a checkplot of the chip is shown in Figure 9-13.

9.4.1. Data path

The DChip data path is 8 bits wide and contains nineteen registers and an ALU connected by a single bus. The ALU performs 8-bit computations and has two input registers for its input operands and two output registers one of which can be chosen to deposit the result of the operation.

Each instruction execution is composed of the two non-overlapping phases of a two-phase clock. During the first phase (φ_1), the bus transfers data from one register to another. One of the registers can be enabled onto the bus and the contents of the bus can then be loaded into a different register. Meanwhile the carry path of the ALU is precharged during this phase of the clock. During the second phase (φ_2), the ALU performs its computation and loads the result into either one of the output registers (ALUO1 or ALUO2). The operands of the computation come from the ALU input registers ALUI1 and ALUI2, which had to have been set up during previous register transfer operations. Double operand operations will hence take at least two instructions to execute. The bus is precharged during the second phase of the clock.

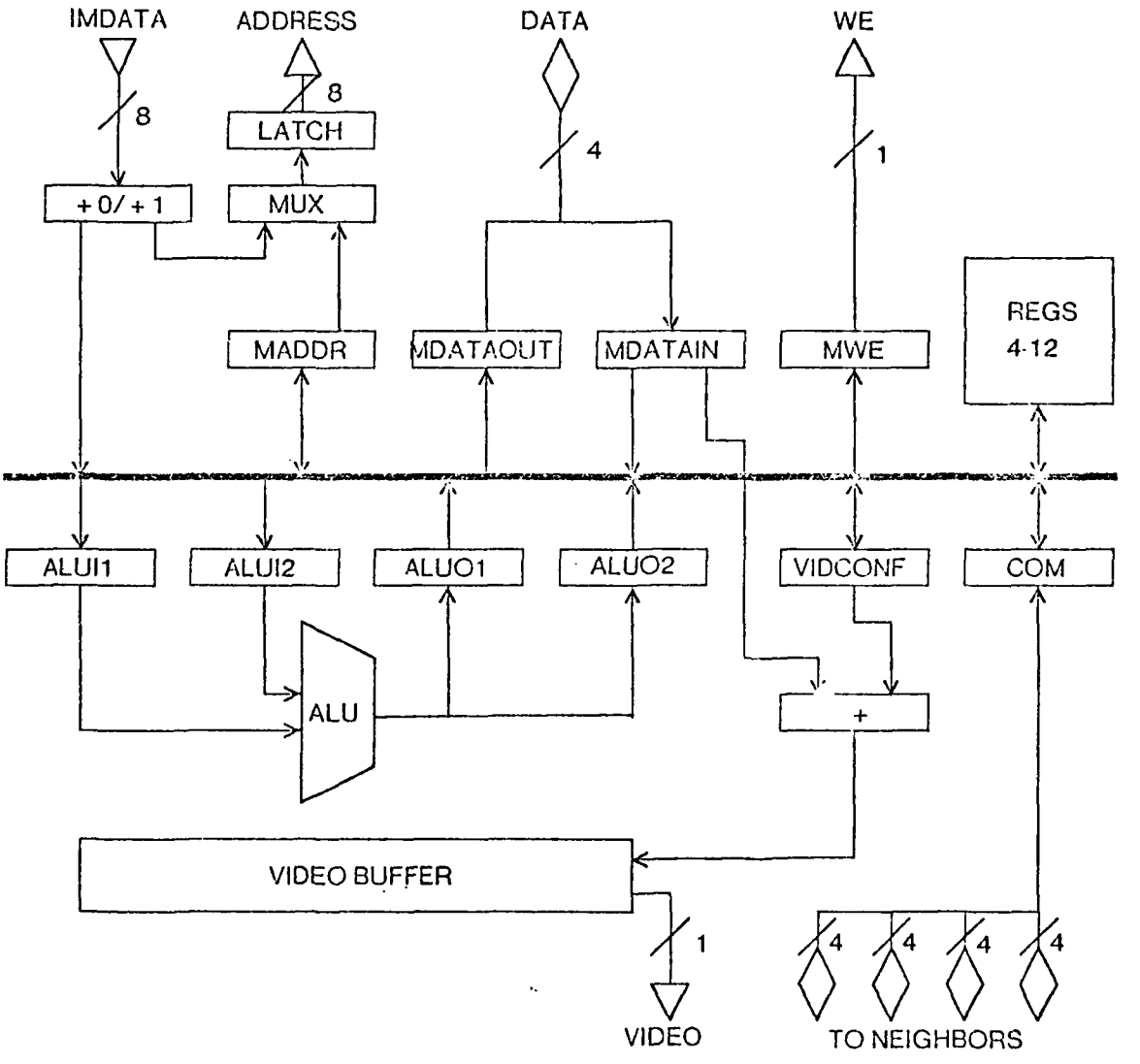


Figure 9-12: Block diagram of *DChip*.

Four registers (MADDR, MDATAIN, MDATAOUT, and MWE) interface the data path with the RAM chips. MADDR can be used to provide an 8-bit address to memory chips, MDATAIN is used to latch the result of a memory read, while MDATAOUT provides the data for a memory write. Only the lower order 4 bits of these registers are used. The lowest order bit of the MWE register is enabled onto the write enable pins of the memory chips during a write operation.

Another register (COM) interfaces the data path to neighboring DChips.

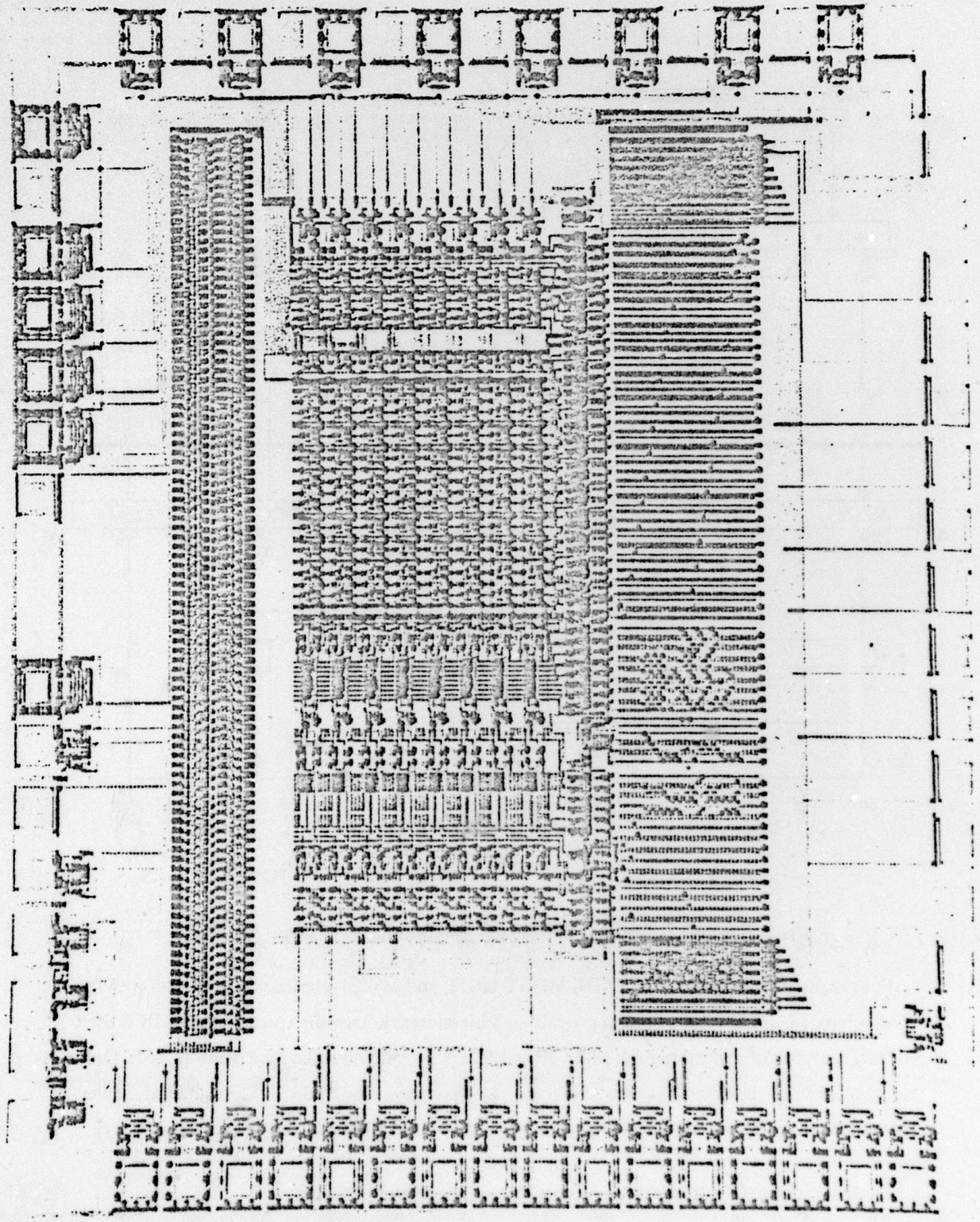


Figure 9-13: Checkplot of *DChip*.

9.4.2. Memory interface

DChip interfaces to the four 64K RAM chips by 13 wires. 8 of these wires (ADDRESS) carry the address and are bussed to all the four memory chips. 4 wires (DATA) are used for the data of the four memory chips. These wires are bidirectional because they are used to both read and write memory data. The 13th wire (WE) is bussed to the write enable pins of all the four memory chips.

The address is generated from either the MADDR register of the data path or the input from the IMDATA pins. The IMDATA passes through an incrementor, the result of which is either the same as the input or incremented by one. The result from the incrementor is then multiplexed with the contents of the MADDR register to the input of the pipelining latch before the ADDRESS wires. The MADDR register is intended to provide computed addresses to memory and is used to provide table look-up operations.

The incrementor and the multiplexer are controlled by the two bits of the ADDRSELECT input. If both the bits are zero, then the value presented at the IMDATA input is available unchanged to the input of the pipelining latch. If either of bits is one, then the IMDATA input is incremented by one and selected to the input of the latch. (Note that the ADDRSELECT inputs act in a manner similar to the *RInc* and *CInc* control pins discussed in Section 9.1.2.) If both of bits are one, then the content of the MADDR register is selected to the input of the latch. The pipelining latch is controlled by the ADDRLATCH input. When this control signal is high the input to the latch is latched and remains latched even when the signal goes low. The output of the latch is always enabled on the ADDRESS pins.

The DATA pins are tri-state pins that can be driven either from the chip or externally from the memory chips. If driven from the chip, they are driven with the contents of the MDATAOUT register. If driven by the memory chips, then the contents can be latched into the MDATAIN register. The two functions are controlled by the DATAENABLE and DATALATCH inputs. When both these inputs are low, the DATA pins are held at high impedance. When DATAENABLE is asserted, the MDATAOUT register is enabled out, and when DATALATCH is asserted the input is latched into the MDATAIN register.

The value of the lowest order bit of the MWE register is enabled onto the WE pin when DATAENABLE is on.

None of the elements in the path of the memory address are controlled by the data path clock; the

memory can hence be addressed and accessed asynchronously. The pipelining latch is intended to speed up the addressing process.

9.4.3. Interprocessor communication

The display chips can communicate with other similar chips to provide the data movement needed to move parts of the image from one part of the display to another. Each display chip has four independent tri-state ports which can be used to interconnect several of these chips into one of several interconnection mechanisms (Figure 9-11). Data from the COM register can be sent to a neighbor and the data from another neighbor can be latched into the same register during each communication cycle, which is controlled by the COMCLOCK. Because the transfer is expected to take place in one COMCLOCK cycle, the COM register is implemented as a Master-Slave register. When the clock is high, the contents of the Slave is enabled out onto one of the four ports and the contents of a different port are latched into the Master part of the register. When the clock goes low, the contents of the Master are transferred to the Slave. The direction of the transfer is controlled by two control inputs COMOUT and COMIN, each of which contains two bits. Each of these controls specify the port onto which the data is to enabled and the port from which the data is to be latched.

The communication timing, similar to memory addressing, is asynchronous to the datapath clock. This is to allow the *Align* operation of BitBlt to be performed fast as possible independent of the speed of the other parts of the system. It also allows the *Align* operation to overlap with memory cycles, a requirement discussed in Section 4.3.4.

9.4.4. Op code

The bus and the ALU are controlled by an eight-bit op-code. During the first phase of processor clock (φ_1), these eight bits specify the source and destination registers for the bus transfer. During the second phase (φ_2), they specify the ALU operation.

The bus transfer takes place by enabling the contents of one of the registers onto the bus and then latching the contents of the bus into another register. As far as the bus transfer is concerned some of the registers in the data path are read-only or write-only. In particular, MDATAOUT is write-only, MDATAIN is read-only, MWE is write-only, ALUI1 and ALUI2 are write-only, and ALUO1 and ALUO2 are read-only. The codes used to specify the source and destinations of the bus transfer are the following.

Code	Source	Destination
0	IMDATA	MWE
1	MADDR	MADDR
2	MDATAIN	MDATAOUT
3	VIDCONF	VIDCONF
4-12	Regs 4-12	Regs 4-12
13	COM	COM
14	ALUO1	ALUI1
15	ALUO2	ALUI2

The ALU operates on two eight-bit operands which come from the two ALU input registers and deposits its result into one of the two ALU output registers. The carry-bit from the most significant bit of the ALU is latched into a carry latch which can be used to control subsequent ALU operations. The ALU op-code is specified in four parts. The first part is a five-bit field which specifies the operation to be performed by the ALU. The second part is a single bit which specifies if the result of the ALU operations is to be half-byte swapped. The third part is a single bit which specifies which of the output registers is to be loaded with the result of the ALU operation. The fourth part is another single bit which specifies if the operation is a conditional. If it is, then the carry from the previous operation is used to abort the loading of the output register. The output register is loaded only if the carry latch is not asserted.

Code	ALU Operation
0	NOP (Output register not loaded)
1	0
2	−1
2	ALUI1
3	ALUI2
4	NOT ALUI1
5	NOT ALUI2
6	−ALUI1
7	−ALUI2
10	ALUI1+1
11	ALUI2+1
12	ALUI1−1
13	ALUI2−1
14	ALUI1+ALUI2
15	ALUI1+ALUI2+Carry
16	ALUI1−ALUI2
17	ALUI1−ALUI2−Borrow
20	ALUI1 AND ALUI2
21	ALUI1 OR ALUI2
22	ALUI1 XOR ALUI2
23	ALUI1 OR ALUI2'

24	ALU1 lshift 1 (Zero Insert)
25	ALU2 lshift 1 (Zero Insert)
26	ALU1 lshift 1 (One Insert)
27	ALU2 lshift 1 (One Insert)
30	ALU1 lrot 1
31	ALU2 lrot 1

9.4.5. Video buffer

There is a single-bit video buffer on each chip. It consists of two shift registers which shift in opposite directions (Figure 9-4). Each shift register is 96 bits long. The shift registers are controlled by four signals. As seen before, DATALATCH latches the contents of the DATA wires into the MDATIN register. VIDSHIFTIN shifts one of the video latch bits into the Input Shift Register. The contents of the VIDCONF register specify which bit gets shifted in; the contents of the two registers are ANDed and the result ORed to produce the bit that gets shifted into video. VIDTRANSFER transfers the contents of the Input Shift Register to the Output Shift Register. *VIDSHIFTOUT* shifts out the top bit of the Output Shift Register to the VIDEO pin. It also enables the VIDEO pin which is otherwise held at tri-state.

9.4.6. Pin summary

DChip has 64 pins, 38 of which are used for data and 23 for control. Three pins are reserved for Power, Ground, and Substrate.

Of the 38 data pins, 16 are used communicate with the four neighbors. Another 16 are required to input the IMDATA, and output the ADDRESS, both of which are eight-bit numbers. The memory DATA wires require 4 pins and the WE wire requires 1 pin. Because this chip contains a single bit video buffer, there is only 1 VIDEO pin.

Of the control pins, 8 are used to input the ALU op code. There are 2 ADDRSELECT pins, 2 each for COMIN and COMOUT, and 1 each for the DATAENABLE, DATALATCH, ADDRLATCH, and VIDTRANSFER inputs. The main clocks for the datapath (φ_1 and φ_2) are input from different pins, while the other three clocks for the neighbor shifting, the input and output shift registers are input from one pin each.

9.5. Simulation and performance

Using the proposed design, this section presents the instruction sequences required in the inner loops for the BitBlt copy and invert operations and makes speculations about their performance. Both these inner loops assume that the masking required for the end conditions is handled elsewhere.

This discussion assumes a processor cycle time of approximately 200 ns. Assuming a memory cycle time of 400 ns, a memory read or write takes 2 processor cycles. Also assumed will be that a neighbor communication step also takes 200 ns. A two dimensional end around shift hence takes a maximum of 4 processor cycles. If we assume the abstraction of the procedures *Align*(x_1,y_1,x_2,y_2), *MemoryRead*(x,y), and *MemoryWrite*(x,y) which take 4, 2, and 2 processor cycles respectively, then the inner loop of the BitBlt copy operation is:

BBCopyLoop:
(1) Align(x_1,y_1,x_2,y_2); MemoryWrite(x_2,y_2); $x_2 := x_2+8$; $y_2 := y_2+8$;

(2)

(3) MemoryRead(x_1,y_1); $x_1 := x_1+8$; $y_1 := y_1+8$;

(4)

(5) MDATAOUT := COM;

(6) COM := MDATAIN; GO TO BBCopyLoop;

The inner loop of the BitBlt copy operation comprises of 6 microinstructions, which at the rate 200 ns/instruction would take 1.2 μs to execute. To scroll a 768x1024 screen would hence take approximately 15ms, which is less than one frame time for a refresh rate of 60 frames/second.

The inner loop of the BitBlt invert operation does not require the *Align* operation if a display region is being inverted in place. The loop is hence the following:

BBInvLoop:
(1) ALUI1 := MDATAIN; MemoryWrite(x,y); x:=x+8; y:=y+8;

(2) ALUO1 := NOT ALUI1;

(3) MDATAOUT := ALUO1; MemoryRead(x,y);

(4) GO TO BBInvLoop;

Using this four instruction inner loop, a 768x1024 display can be inverted in place in approximately 10 ms!

9.6. Design scalability

This section the implications of scaling various parameters to the display chip design. These parameters are -

- The size of the square of the memory organization. The size of the square determines the number of pixels that can be updated in parallel.
- The size of each RAM chip. The size of the memory chip determines the total size of the display for any given number of memory chips.
- The number of gray-scale bits in each pixel. This value determines the number of different shades portrayable by each point.

The total number of memory chips in a display system using the MxM square memory organization with g bits of gray-scale system is gM^2. If each chip contains k bits then the total number of bits in the system is gkM^2.

The display chip contains 6 external pixel data paths, four for the neighbors, and one each for the memory and the video (the current design only has one bit of video). It also contains 2 external address paths (for IMDATA and ADDRESS), where each address is $\ln(k)/2$ bits. Each display chip hence uses $6g+\ln(k)/2$ pins which will change with g and k.

Scaling the size of the square affects the number of pixels that can be udpated simultaneously. However, if the desire is to build a display with a given total area, then the size of individual memory chips will change with a change in the size of the update square. If, for example, the size of the update square is doubled (i.e. $M := 2M$), then the memory chips can be 1/4th the size resulting in 2 fewer pins. Scaling the size of the memory chips has the converse effect. The biggest impact on the display chip is by the number of bits in the gray-scale because of the need of 6 pins per bit in gray-scale. In fact, as g increases beyond a certain point, it is indeed not possible to handle the whole pixel in one display chip. A solution to this would be split the pixel into a number of bit sliced display chips. This solution results in slower speeds for operations w' ch have to propogate a carry through the entire pixel value.

9.7. Status

DChip was fabricated using the MOSIS system provided by ISI. The chip was sent for fabrication on October 31, 1981. Seven parts were returned on January 6, 1982. They were tested functionally using an elementary test rig and were found to be working functionally. They were not tested for timing and speed information.

This thesis does not address the design of the controller which would control the display memory system. The controller design is as crucial as the design of the memory system, because it is up to the controller to fully utilize the capabilities of the memory system.

Chapter 10
Conclusion

This thesis is the study of the feasibility of a display design in which the underlying primitive is the parallel update of any small square region of the display. This primitive operation is justified by the well known *principle of locality*, which states that if any display operation updates a given pixel then the next pixel it updates will be in the close proximity of the current location. Since the display is inherently a two-dimensional device, these successive pixels are likely to be in a close two-dimensional neighborhood, and can hence be updated in parallel using the square update primitive. The conventional scan line approach allows the parallel update of only horizontal spans and is hence convenient only for operations in which successively updated pixels lie along scan lines. This thesis attempts to demonstrate that square updates are more desirable. It does so by looking at a wide range of display applications and showing algorithms that can be implemented efficiently using square updates.

The square organization is effective only if the size of the square being updated is approximately equal to the size of the display objects. If the size of the objects is much larger than the size of square than the scan line organization can achieve the same efficiency as the square organization (efficiency is measured in terms of the number of pixels actually updated as a percentage of the number that can be updated). So, for example, the scan line organization is as efficient as the square organization to scroll the contents of the whole display. Conversely, the 8x8 square organization is more efficient than the 64x1 scan line organization to update an 8x10 character. Since most operations call for the update of small squarish objects, the square organization is indeed a justified choice for the display organization. [Sutherland 81] contains a more detailed performance model justifying the advantages of the 8x8 memory organization.

The *BitBlt* operation (defined in Chapter 4) can be implemented more efficiently for the square display organization than the scan line organization. This operation has been found useful for a large class of display applications which operate on rectangular regions. It can also be used for non-

rectangular operations like drawing lines and filling polygons. This is done by precomputing segments of all possible lines and polygons and putting them together to form the desired images. This is a very powerful approach to achieve high performance for these operations and eliminates the need for high speed processing. The algorithms which show that strokes can indeed be used for graphics are the largest contribution of this research. These algorithms can be extended for gray-scale graphics to produce smooth edges.

The stroke algorithms can be adapted to display lines and polygons whose endpoints do not lie on pixel centers. Non-integer endpoints are necessary to avoid dynamic aliasing, which causes jitter in a moving image because of the quantization of endpoints to pixel centers. But the number of strokes required when the endpoint restriction is removed is impractically large. Under these circumstances, a parallel processing approach can be used to compute each stroke. The endpoints of lines and polygons with non-integer endponts require a square array of processors, the size of the square being exactly the same as the size of the square which can be updated by the display architecture.

The square display architecture is also ideal for most low-level image processing algorithms. Chapter 8 discusses a given class of such algorithms, which are commonly used to present images in more desirable formats. The attempt of this presentation was not to discover new image processing algorithms, but to show that existing algorithms can be easily adapted to provide high performance using the square array of processors provided by our display design.

Chapter 9 presents the design of a simple processor, an array of which can be used to implement the algorithms discussed for the various different applications. The fact that an extremely simple processor is sufficient for a very large class of applications is, in my view, the single most significant contribution of this thesis.

10.1. Future designs

The design of the display chip, although simple, uses 64 pins. This can be easily reduced if the memory chips can be modified slightly. The easiest and most beneficial modification would be allow the address incrementing required to access squares which overlap word boundaries. One extra pin in the memory chip, which when asserted would increment the address by one, is sufficient to decrease the pin count in the display chip by 18. The other desirable modification is to provide shift registers which can store all the bits read by the sense amplifiers when one row of the memory array is enabled. This data can be used serially for screen refresh, which would both increase the total memory bandwidth available and decrease the pin count of the display chip.

The idea of using a square array of processors to provide the ability of updating several pixels in parallel can be extended to encompass a much larger range of applications than the ones discussed in this thesis. In the area of computer graphics this thesis restricted itself to straight lines and edges. Similar algorithms can be used for parametric curves, but they require better computing primitives in each display chip.

And lastly, I would like to emphasize the need to build a display to really study the performance tradeoffs. Theoretical models and simulations provide a lot of information but are not sufficient for two reasons. The first reason is that the real life display usually presents surprises that the simulations did not account for. But the second and more important reason is that the use of a display gives rise to new and innovative uses of computer graphics, and these could completely change the performance tradeoffs that the display design had assumed.

References

[Ball 81] Ball, E.
Canvas: the Graphics Package for the Spice Personal Timesharing System.
In *Computer Graphics 81: Proceedings of the International Conference*, pages 269-280. 1981.

[Baskett 76] Baskett, Forrest and Shustek, Leonard.
The Design of a Low Cost Video Graphics Terminal.
Computer Graphics 10(2):235-240, Summer, 1976.

[Bawden 77] Bawden, A., et.al.
Lisp Machine Project Report.
Technical Report AIM 444, M.I.T. A.I. Lab, Cambridge, Mass., August, 1977.

[Bechtolsheim 80] Bechtolsheim, Andreas and Baskett, Forest.
High-Performance Raster Graphics for Microcomputer Systems.
Computer Graphics 14(3), July, 1980.

[Bresenham 65] Bresenham, J.E.
Algorithm for computer control of a digital plotter.
IBM Systems Journal 4(1):25-30, July, 1965.

[Clark 80] Clark, J.H., and Hannah, M.R.
A High-Performance Smart Image Memory.
Lambda, 3rd Quarter, 1980.

[Crow 76] Crow, F.C.
The Aliasing Problem in Computer-synthesized Shaded Images.
PhD thesis, University of Utah, March, 1976.

[Denes 75] Denes, P.B.
A Scan-type Graphics System for Interactive Computing.
Proceedings IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structures:21, May, 1975.

[Feibush 80] Feibush, Eliot A., Levoy, Mark, and Cook, Robert L.
Synthetic Texturing Using Digital Filters.
Computer Graphics 14(3):294-301, July, 1980.

[Jordan 74] Jordan, B.W., Jr. and Barrett, R.C.
A Cell Organized Raster Display for Line Drawings.
CACM 17(2):676, February, 1974.

[Kajiya 75] Kajiya James T., Sutherland Ivan E., and Cheadle Edward C.
A Random-Access Video Frame Buffer.
Proceedings IEEE Conference on Computer Graphics, Pattern Recognition, and Data Structures:1, May, 1975.

[Leler 80] Leler, William J.
Human Vision, Anti-aliasing, and the Cheap 4000 Line Display.
Computer Graphics 14(3):308-313, July, 1980.

[Lucas 81] Lucas, Bruce.
Personal communication.

[McCracken 75] McCracken T.E., Sherman B.W., Dwyer S.J. III.
An Economical Tonal Display for Interactive Graphics and Image Analysis Data.
Computers & Graphics 1(1):79-94, 1975.

[Noll 71] Noll, A. Michael.
Scanned-Display Computer Graphics.
CACM 14(3), March, 1971.

[Ophir 68] Ophir D., Rankovitz S., Shepherd B.J., and Spinard R.J.
BRAD : The Brookhaven Raster Display.
CACM 11(6):415, June, 1968.

[Roberts 65] Roberts L.G.
Machine Perception in Three-dimensional Solids.
Optical and Electrooptical Information Processing. MIT Press, Cambridge, Massachusetts. :159-165, 1965.

[Rothstein 79] Rothstein, J. and Weiman, C.F.R.
Parallel and sequential specifiction of a context sensitive language for straight lines on grids.
Computer Graphics and Image Processing 5, 1979.

[Sproull 79] Sproull, Robert F.
Raster Graphics for Interactive Programming Environments.
Computer Graphics 13(2):83-93, August, 1979.

[Sproull 81] Sproull Robert F.
Using Program Transformations to Derive Line-Drawing Algorithms.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1981.

[Sutherland 70] Sutherland, Ivan E.
Computer Display.
Scientific American , June, 1970.

[Sutherland 81] Sproull R.F., Sutherland, I.E., Thompson A., Gupta, S., and Minter, C.
The 8 by 8 Display.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1981.

[Terlet 67] Terlet, J.H.
The CRT display subsystem of the IBM 1500 instructional system.
AFIPS Conference Proceedings 31, FJCC, 1967.

[Thacker 81] Thacker, C.P., McCreight, E.M., Lampson, B.W., Sproull, R.F., and Boggs, D.R.
Alto : A Personal Computer.
In Siewiorek, D., Bell, C.G., and Newell, A. (editor), *Computer Structures : Principles and Examples, second edition*, . McGraw-Hill, 1981.

[Warnock 80] Warnock John E.
The Display of Characters Using Gray Level Sample Arrays.
Computer Graphics 14(3):302-307, July, 1980.

[Weiman 80] Weiman, Carl F.R.
Continuous Anti-Aliased Rotation and Zoom for Raster Images.
Computer Graphics 14(3), July, 1980.